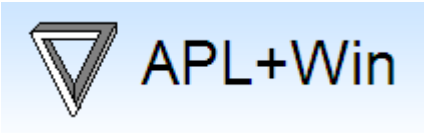
APL+Win with R



Client/Server Integration by Ajay Askoolum

http://www.apl2000.com

http://www.r-project.org/

This paper demonstrates the basic techniques for client/server integration of APL+Win version 11.1 and R version 2.14.2 (32-bit) for the Windows platform.

R is a programming language for statistical computing and data visualisation. It consists of a language plus a run-time environment with graphics, a debugger, a number of pre-defined system functions, and the ability to run programmes stored in script files. R is also extensible via the installation of *packages*—read libraries; there are many packages available from a central R repository, as well as others from commercial providers. Unlike APL+Win, which is proprietary software, R is free software that is subject to a GNU public license.

By integration, I mean using APL+Win as a client to R as server and using R as a client to APL+Win as server. The technology in question is that of OLE automation, that is, the OLE/COM (Object Linking and Embedding/Component Object Model) interface. This enables one more option for collaborative application development.

- The APL+Win developer gains access to the rich statistical analysis and data visualisation capability that R offers.
- The R developer gains access to the ultimate 'package', APL+Win: it can solve or simplify solutions to problems as yet unthought-of, including the provision of a graphical user interface for an R application.

Table of Contents

1. Introduction

R is an interactive cross-platform development environment that has robust code and data distribution functionality. R's expressive power enables you to do nearly anything, *as long as you know how to*. One of the key reasons for R's success is that there is a lot of goodwill[1] and support freely available from the network—http://groups.google.com/group/r-help-archive/topics—of other R users, referred to as the R mailing list.

```
> print(R.version.string);
[1] "R version 2.14.2 (2012-02-29)"
> print(win.version());
[1] "Windows XP (build 2600) Service Pack 3"
```

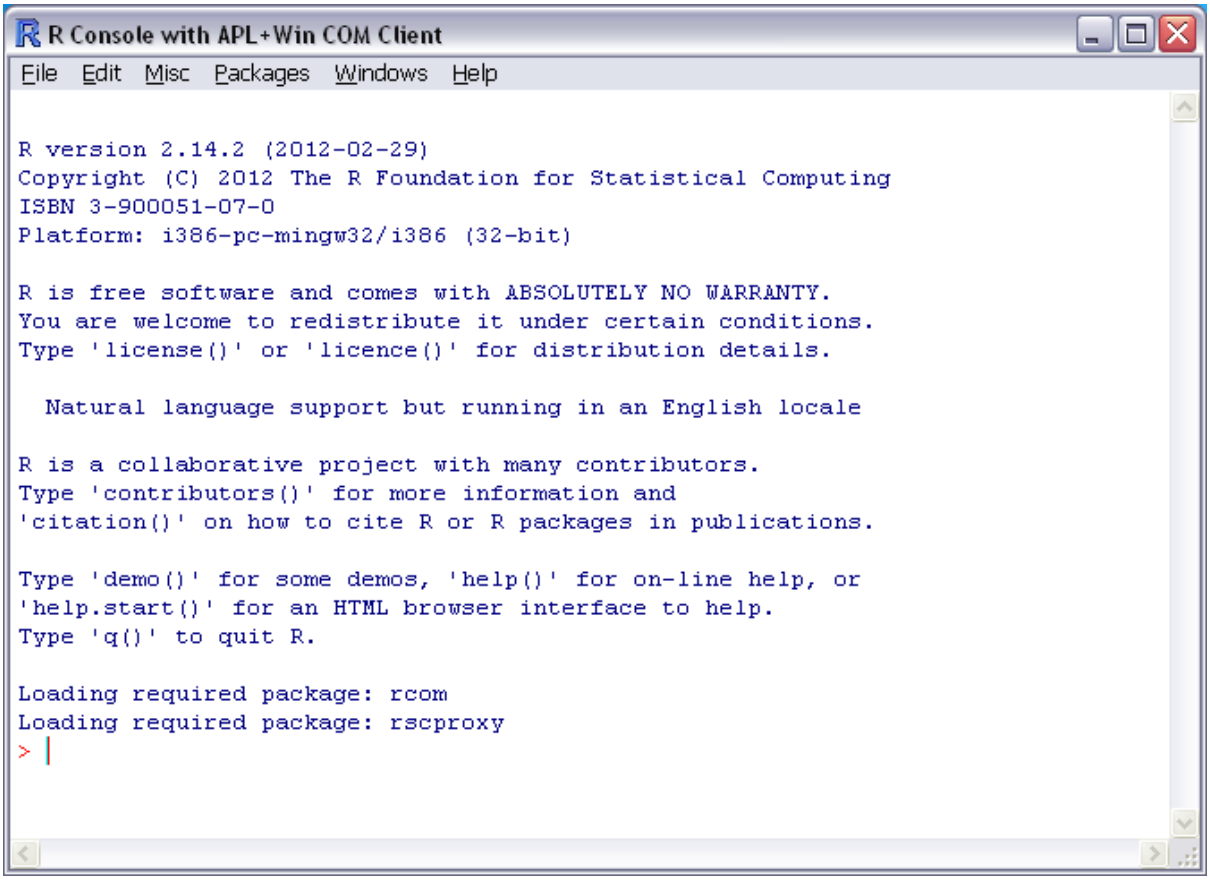
R is a powerful personal computing language that is highly productive in the hands of subject matter experts, especially those involved in the analysis and visualisation of data. Its use is widespread among statisticians, actuaries, clinical researchers, science and mathematics academics, and their students.

Like any well-loved tool, R creates its own comfort zone and tends to be used for disparate associated tasks, such as data acquisition, generation, and cleaning, which arise routinely in the context of data analysis and visualisation.

Although R is a self-contained environment, it is of interest to APL+Win developers because of the availability of packages that enable it to be embedded in an APL+Win application—R can also embed APL+Win—using DCOM/COM (Distributed/Component Object Model) technology.

1.1. R Features out of the box

By 'out of the box' I mean a set-up as described in Installation. In other words, the starting point is R version 2.14.2 with libraries rcom and rscproxy already installed.



[1] Nonetheless, be prepared for terse (condescending?) responses such as ?*function*, read the FAQ, see *reference* etc.

R is a cross-platform development tool. Therefore, it also incorporates a number of features that consolidate access to platform resources in a standard way.

- This maintains the portability of R code.
- Reduces the reliance on platform specific resources, such as Windows API calls.

My installation of R has the following:

```
> search()
 [1] ".GlobalEnv"        "package:stats"     "package:graphics"
 [4] "package:grDevices" "package:datasets"  "package:rcom"
 [7] "package:rscproxy"  "package:utils"     "package:methods"
[10] "Autoloads"         "package:base"
```

In all, this presents a large number of objects to investigate and assimilate.

```
> sum(sapply(sapply(search(),objects),length))
[1] 2449
> sapply(search(),objects) # Use this to enumerate the objects in each package
```

This number of objects translates into a learning curve that is simply formidable. However, R is a language of pathways: there are many ways of achieving the same result, more or less.

- There is overlap among the objects; therefore, the objective is to find the solution that solves the problem to hand and to investigate further only when the solution is found lacking. The learning curve is tackled on a need to learn basis.
- The learning curve is incremental; as with APL+Win, you can start with the basics and as fluency increases, move on to explore the power of R.

1.1.1. R Pathways

As its prime focus, R emphasises the need to accomplish rather than optimize the programming task to hand. It is the solution rather than its speed that is of the essence. Hence, R is the ultimate personal computing tool; it blurs the distinction between users and programmers almost completely. Users are the programmers.

R is not a 'point and click' environment; it has no graphical user interface building capability and there are no commercial services whom you can lobby—or indeed pay—for features and functionality[2]. This explains to some extent the emergence of overlapping packages: if it is not available, write your own albeit subject to the constraint that you must comply with laid down standards. R packages tend to be complementary and written for own use. Then, what better way is there to have the package debugged or enhanced than by making it freely available?

The opposite scenario—applicable to the commercial world—is to develop packages that evolve without overlaps with other packages; missing features are introduced on a demand-driven basis and the functionality is re-factored in order to gain memory and speed efficiencies. R's stance is to allow technology advances to address such constraints. Commercial packages tend to be self contained; that is, they replace rather than complement other packages.

R is rich in pathways—the interconnection of available objects to yield a result. Therefore, it is quite easy to find a worked example for any given problem, if the result is all that is sought. However, pathways have implications relating to the efficiency—measured in terms of speed and resource usage—of a given solution. R is itself a solution and there is some disquiet relating to its speed and resource usage.

Pathways make the learning curve steeper as there are many syntax patterns to learn and they also introduce the risk of name conflicts. Loading packages diminishes the amount of memory available for personal computing, increases the prospects of name collisions, and generally makes the investigation of solutions more strenuous.

An illustration of pathways: What date is today less two years?

```
> myDate<-Sys.Date();
> print(myDate);
```

[2] One notable exception is Revolution Analytics, see http://www.revolutionanalytics.com/products/revolution-enterprise.php.

```
[1] "2012-02-25"
> # Pathway 1
> print(seq(myDate,length=2,by="-2 years")[2]);
[1] "2010-02-25"
> # Pathway 2
> thisDate<-as.POSIXlt(myDate);
> thisDate$year<-thisDate$year-2;
> print(thisDate);
[1] "2010-02-25 UTC"
> # Pathway 3
> library(lubridate);        # load library
Overriding + and - methods for POSIXt, Date and difftime
> print(myDate-years(2));
[1] "2010-02-25"
> detach(package:lubridate); # unload library
```

Paradoxically, both the key strengths and weaknesses of this language arise from a single feature: the number of pathways it offers to achieve the same end result. If anything, this aspect of the language worsens, as you acquire greater fluency, because R is an extensible language. The addition of packages introduces new keywords and, quite often, there are subtle differences in the end result depending on the solution's implementation.

Note that the result of each pathway may require some scrutiny: the first pathway requires indexing as it is the second element of the output that is the result sought, the second has the suffix UTC (Universal Coordinated Time) although this is meaningless without time values, and the third carries an overhead that may be detrimental. Let me illustrate how the name conflict, alluded to earlier, arises.

```
> # Pathway 2 from above
> d<-as.POSIXlt(myDate);
> d$year<-d$year-2;
> print(d);
[1] "2010-02-25 UTC"
> # Pathway 3
> library(lubridate);        # load library

Attaching package: 'lubridate'

The following object(s) are masked _by_ '.GlobalEnv':

    d

> print(myDate-years(2));
[1] "2010-02-25"
> detach(package:lubridate); # unload library
```

By simply renaming the identifier in the second pathway, I introduced a name collision. This sort of scenario simply makes life harder as it cannot be anticipated.

R does not have facilities for producing customised graphical user interfaces. Therefore, applications are run either from the command line or via script files without user interaction in the conventional sense.

1.1.2. R Anything goes, if it works!

A further feature of R that complicates matters is that everything is an object and there are no invalid object names. Functions are used as arguments. In some contexts, inadvertently or by design, functions and variables can be given names that correspond to the keywords in the language or its packages, thereby masking the original definition.

Therefore, it is impossible to prescribe a naming convention such as Pascal case—where composite words are used without hyphenation or punctuation and the initial letter of every word is capitalised–or Camel case—as Pascal case but the initial letter of the first word is in lowercase. Nevertheless, with due diligence, try to use the Pascal and Camel naming conventions for functions and other identifiers respectively. This is a short illustration of the foregoing statements.

```
> # ------------------------------------------------------------------------
> # This is a script file history.R, copied and pasted into an R session
> # Lines from the script file are shown in red, R's response are in blue
> # ------------------------------------------------------------------------
> # Define a function named PascalCase that returns its argument squared
> PascalCase<-function(arg){
+ return (arg*2);
```

```
+ }
> # Define an identifier named camelCase as a vector
> camelCase<-c(8,2.3);
> # See the value of camelCase
> camelCase
[1] 8.0 2.3
> #Call PascalCase with argument camelCase
> PascalCase(camelCase)
[1] 16.0  4.6
> # Any name for an identifier - note the quotes
> '123'<-c(7,2,3);
> # see its value
> get('123') # Must use the keyword get
[1] 7 2 3
> # else use backquote (`)
> `123`<-c(7,2,3);
> `123`+10;
[1] 17 12 13
> # Any name for a function - beginning with a digit
> '15f'<-function(arg){
+ return(arg+pi);
+ }
> '15f'(10.0); # expect 10 plus constant pi as the result
[1] 13.14159
> # Demonstration - a function as an argument to another function (sapply)
> sapply(c(1,2,3),FUN='15f') # Expect pi plus vector 1 2 3
[1] 4.141593 5.141593 6.141593
> # Demonstration - a function as an argument to another function (sapply)
> sapply(90,FUN='PascalCase');  # Expect 90 times 2
[1] 180
> # c is a keyword - it returns a vector having concatenated its arguments
> # Example - all integers between 1 and 10 inclusive
> c(1:10);
 [1]  1  2  3  4  5  6  7  8  9 10
> # Let's re-define c as a variable
> c=9.34;
> # See its value
> c
[1] 9.34
> # What happened to Concatenate?
> c(1:10);
 [1]  1  2  3  4  5  6  7  8  9 10
> # Confused?
> ls(); # List of user-defined objects in current session.
[1] "123"        "15f"        "c"          "camelCase"  "d"
[6] "myDate"     "PascalCase" "thisDate"
> # Like-named functions can be confusing e.g.
> # c and C for concatenate and Contrasts (of a factor)
> # I (often used as a counter variable) is a function, stands for AsIs
> # T and F are predefined variables having the expected values of TRUE and FALSE
> # Therefore, T can be used in place of TRUE and F in place of FALSE ... but
> T<-FALSE;
> F<-TRUE;
> # The meanings are now reversed!
```

1.2. R Language structural features

From the foregoing short demonstration of some of R's peculiarities—which you should avoid—some structural features of R become obvious; they provide a bare bone guideline for R programming.

- R is an interpreted language; it supports environments—read 'scope' or 'namespaces' depending on context—and can save its sessions as workspaces.
- R is case sensitive.
- It uses index origin 1 for indexing; there is no other option.
- Single-quote (') and double-quote (") can be used interchangeably
- <- and = can be used interchangeably for assignment; comparison is == (equal to) or != (not equal to). The keyword *identical*(arg1,arg2) is the direct equivalent of APL+Win match (Ξ). Take care to enclose negative numbers in round brackets for comparison e.g. R < (-3); otherwise the code becomes ambiguous. One exception is when calling a function with named arguments; then, you must use = and not <-.
- # marks the start of an in-line comment

- By convention, { must be the last character on a line; it marks the beginning of a scope. I prefer the C# convention where the opening and closing double brace characters appear on a line by themselves; this makes for clearer indentation of code.
- By convention,} must also be the last character on a line; it marks the end of a scope.
- All identifiers between {} are local in so far as it is not possible to define a global variable within a function; however, with lexical scoping, an identifier in the global scope can be used.
- Semi-colon (;) marks the end of a statement; although it is optional, it is advisable to use it always for the sake of clarity.
- The keyword *return* is optional. By default, an R function will return the last value assigned in its scope. It is advisable to use the keyword always for clarity.
- In command line mode, + denotes a continuation line. R statements can span several lines: the interpreter senses when a statement is complete or accepts the end of statement marker.. Press Esc to abort the prompt for completing an expression.
- \ denotes the start of an escape sequence; use / or \\ where \ is required, as in a path name. \r denotes carriage return, \n denotes new line, \t denotes tab etc. Try cat("This\nis\nan\nexample.") to see the effect.
- R evaluates complete statements and returns their results, if any, immediately; an error is returned when a statement is complete but invalid. In other words, it works exactly like APL+Win in interactive mode.
- > is the default command line prompt.
- + is the prompt for continuation lines.
- Press the ESC key to exit the line continuation prompt, where necessary.

1.2.1. R Session Attributes

A number of features of the R interface can be customised. The following enumerates the list of features:

```
as.matrix(options(),ncol=2)
```

Options are queried using

```
> getOption("digits") # Equivalent of print precision
[1] 7
> options()["width"]  # Print width
 [1] 80
> options()$papersize # PDF or Postscript output
[1] "a4"
```

And, they can be set thus:

```
> options(digits=9)
```

The theme of pathways continues: another way to query an option attribute is:

```
> getOption("digits") # Equivalent of print precision
[1] 9
```

1.2.2. Globalisation tokens

There are token gestures to globalization, see below, but dates are expected in yyyy-mm-dd format by default.

```
> options("OutDec")  # Decimal ... Globalisation token
 [1] "."

> Sys.getlocale()
[1] "LC_COLLATE=English_United Kingdom.1252;LC_CTYPE=English_United
Kingdom.1252;LC_MONETARY=English_United Kingdom.1252;LC_NUMERIC=C;LC_TIME=English_United
Kingdom.1252"
> Sys.timezone()
[1] "GMT"
```

1.2.3. Environment Variables

R has access to Windows environment variables.

```
> Sys.setenv(newAjay="c:/temp/") # Create a new variable
> Sys.getenv("newAjay")          # Get its value
```

```
[1] "c:/temp/"
> Sys.setenv(newAjay="")          # Delete an existing environment variable
> Sys.getenv("windir")            # Get the system directory
[1] "C:\\WINDOWS"
```

1.3. R Developer hints

Developers accustomed to Integrated Development Environments (IDE) that have auto completion or intellisense—a feature that provide hints about the construction of the current line of code—may find the R command line rather intimidating at first. However, R provides a number of features that eases the pain.

\

The backslash character denotes the start of an escape sequence; use / or \\ instead and R will interpret it correctly. However, the APL+Win server will not: it will need to change it to \.

str()

Fully describes R Objects. For example:

```
>  a<-as.matrix(c(seq(1:10),dim(c(2,5))))
> comment(a)<-"An arbitrary variable"
> str(a)
 int [1:10, 1] 1 2 3 4 5 6 7 8 9 10
 - attr(*, "comment")= chr "An arbitrary variable"
```

search()

Enumerates the list of packages already loaded.

```
> search()
 [1] ".GlobalEnv"        "package:stats"     "package:graphics"
 [4] "package:grDevices" "package:datasets"  "package:rcom"
 [7] "package:rscproxy"  "package:utils"     "package:methods"
[10] "Autoloads"         "package:base"
```

xxx{tab} {tab}

Lists the keywords available that begin with *xxx*.

```
> as.character
as.character                as.character.condition
as.character.Date           as.character.default
as.character.error          as.character.factor
as.character.hexmode        as.character.numeric_version
as.character.octmode        as.character.POSIXt
as.character.srcref
```

?keyword or help(keyword)

Invokes the help available on *keyword* in a separate window.

```
?data.frame or help(data.frame)
```

Function

Typing a function name without () lists that function. A large number of R functions are defined in R itself.

```
> rownames
function (x, do.NULL = TRUE, prefix = "row")
{
    dn <- dimnames(x)
    if (!is.null(dn[[1L]]))
        dn[[1L]]
    else {
        nr <- NROW(x)
        if (do.NULL)
            NULL
        else if (nr > 0L)
            paste(prefix, seq_len(nr), sep = "")
        else character()
    }
}
<bytecode: 02336AE8>
<environment: namespace:base>
```

ls()

Enumerates the contents of the current session.

```
> ls()
[1] "a"
```

rm() or remove()

Expunges the arguments from the current session; raises a warning if any of the arguments does not exist.

```
> rm(a,b)
```

apropos()

Finds all objects which contain the argument as a substring.

```
> apropos("union")
[1] ".__C__ClassUnionRepresentation" "isClassUnion"
[3] "setClassUnion"                  "ts.union"
[5] "union"
```

comment()

Adds an arbitrary comment to any object.

```
> comment(a)<-"An arbitrary variable"
```

Profile of a function using several primitives.

Querying attributes of functions.

```
> aoc<-function(radius=5){return(pi * radius^2);}
> comment(aoc)<-"Calculate area of a circle given its radius"
> formals(aoc)
$radius
[1] 5
> names(formals(aoc))
[1] "radius"
> args(aoc);
function (radius = 5)
> body(aoc); # Note position of double braces.
{
    return(pi * radius^2)
}
> str(aoc);
function (radius = 5)
 - attr(*, "srcref")=Class 'srcref'  atomic [1:8] 1 6 1 47 6 47
1 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile'
<environment: 0x01cce2e8>
 - attr(*, "comment")= chr "Calculate area of a circle given its
radius"
> aoc()
[1] 78.53982
```

1.3.1. R Session metrics

A number of metrics are necessary for managing R optimally.

Memory

```
> memory.limit()     # Query available memory
[1] 1535
> memory.limit(2000) # Allocate 2000 MB
[1] 2000
> memory.limit()     # Check Allocation
[1] 2000
> gc()               # Force garbage Collection
         used (Mb) gc trigger (Mb) max used (Mb)
Ncells 186466  5.0     407500 10.9   350000  9.4
Vcells 174328  1.4     905753  7.0   786254  6.0
> memory.size() # Memory currently in use (MB)
[1] 13.63
```

Object Class

```
> as.matrix((sapply(.GlobalEnv, typeof)),ncol=1) # Object Class
           [,1]
as         "double"
allObjects "closure"
abc        "integer"
```

Object Size

```
> as.matrix(sapply(sapply(ls(.GlobalEnv),get),object.size),ncol=1) # Object Bytes
           [,1]
abc         424
allObjects 2828
as           56
```

Timing

```
> data(mtcars)
> system.time(capture.output(mtcars)) # Timing for capture.output(mtcars)
   user  system elapsed
   0.02    0.00    0.07
```

Object Exists

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

1.3.1.1. Object type: query and coercion

R has a large number of functions for querying and coercing its variables/data structures in its default Installation as described below. The prefix * in a name indicates that both the 'is' and 'as' counterparts to the function name exists; otherwise the actual prefix is shown. It is probably advisable to explore this set of functions on a 'need-to'[3] basis since they relate to either very advanced R usage or its interfacing with other languages like C or FORTRAN.

```
*.array                 as.array.default              *.call
*.character             as.character.condition        as.character.Date
as.character.default    as.character.error            as.character.factor
as.character.hexmode    as.character.numeric_version  as.character.octmode
as.character.POSIXt     as.character.srcref           *.complex
*.data.frame            as.data.frame.array           as.data.frame.AsIs
as.data.frame.character as.data.frame.complex         as.data.frame.data.frame
as.data.frame.Date      as.data.frame.default         as.data.frame.difftime
as.data.frame.factor    as.data.frame.integer         as.data.frame.list
as.data.frame.logical   as.data.frame.matrix          as.data.frame.model.matrix
as.data.frame.numeric   as.data.frame.numeric_version as.data.frame.ordered
as.data.frame.POSIXct   as.data.frame.POSIXlt         as.data.frame.raw
as.data.frame.rcomdata  as.data.frame.table           as.data.frame.ts
as.data.frame.vector    as.dendrogram                 as.difftime
as.dist                 *.double                      as.double.difftime
as.double.POSIXlt       *.environment                 *.expression
as.expression.default   *.factor                      as.formula
*.function              as.function.default           as.graphicsAnnot
as.hclust               as.hexmode                    *.integer
*.list                  as.list.data.frame            as.list.Date
as.list.default         as.list.environment           as.list.factor
as.list.function        as.list.numeric_version       as.list.POSIXct
*.logical               as.logical.factor             *.matrix
*.name                  *.null                        as.null.default
*.numeric               *.numeric_version             as.octmode
*.ordered               *.package_version             *.pairlist
as.person               as.personList                 *.raster
*.raw                   *.real                        *.relistable
as.roman                *.single                      as.single.default
*.stepfun               *.symbol                      *.table
as.table.default        *.ts                          as.vector
as.Date                 as.POSIX

You can also query various other characteristics of the active session, especially when
debugging a stubborn bug:

> sessionInfo()
R version 2.14.2 (2012-02-29)
Platform: i386-pc-mingw32/i386 (32-bit)

locale:
[1] LC_COLLATE=English_United Kingdom.1252
[2] LC_CTYPE=English_United Kingdom.1252
[3] LC_MONETARY=English_United Kingdom.1252
[4] LC_NUMERIC=C
[5] LC_TIME=English_United Kingdom.1252

attached base packages:
[1] stats     graphics  grDevices datasets  utils     methods   base

other attached packages:
[1] rcom_2.2-3.1.1 rscproxy_1.3-1

loaded via a namespace (and not attached):
[1] tools_2.14.2
```

1.4. R Error handling, debugging & control structures

R has some error handling and debugging capability and has control structures like any other language.

Refer to the language manuals for details: these topics are perhaps best approached on a #need-to' basis.

[3] Usually, when debugging code.

1.5. Interaction with the filing system

R has a full range of file manipulation functions. In order to query the mode and options of usage, use:

```
>  ?file.[function];
```

Note that these functions promote platform independence; they rely on low level calls into the operating system to deliver code that behaved uniformly across platforms.

file.access(file)	Returns the file access matrix.
file.append(file1, file2)	Concatenates files.
file.choose()	Displays a file open dialogue. This command is highly configurable for filtering of files by name or extension.
file.create(file)	Creates the file or truncates it if it exists already.
file.edit(file)	Opens the file in a new R window for editing.
file.exists(file)	Returns TRUE or FALSE depending on whether the file exists.

file.info(file)	Provides information on the file.
	`> file.info("c:/boot.ini")$isdir;`
	`FALSE`

file.link(from, to)	

file.path(…,fsep)	Assembles a file name.
	`> file.path("c:","ajay","RSTATS","SCRIPTS","AJAY.R",`
	`fsep=.Platform$file.sep)`
	`[1] "c:/ajay/RSTATS/SCRIPTS/AJAY.R"`

file.remove(file)	Delete the file.
file.rename (from, to)	Self-explanatory.
file.show(file)	Opens the file in a new R window.
file.symlink	

1.5.1. tempdir()

This function returns the fully qualified temporary directory path; the result can be used directly for processes in R—note the double\\ --but for APL+Win. Also, you need to add the trailing \\.

```
> tempdir()
[1] "C:\\DOCUME~1\\AJAYAS~1\\LOCALS~1\\Temp\\Rtmp2f270T"
```

For R:

```
> paste(tempdir(),"\\","myfile.txt",sep="");
[1] "C:\\DOCUME~1\\AJAYAS~1\\LOCALS~1\\Temp\\Rtmp2f270T\\myfile.txt"
> gsub("\\\\","/",filename)
[1] "C:/DOCUME~1/AJAYAS~1/LOCALS~1/Temp/Rtmp2f270T/filebcc3ad439e0.log"
```

For APL+Win a simple expression can change the R path separator into a conventional one:

```
      R←'C:\\DOCUME~1\\AJAYAS~1\\LOCALS~1\\Temp\\Rtmp2f270T\\myfile.txt'
      r←'C:/DOCUME~1/AJAYAS~1/LOCALS~1/Temp/Rtmp2f270T/myfile.txt'
      ⍕1↓∊(('\/'⍳R)=2+⎕io)⊂R),¨'\'
C:\DOCUME~1\AJAYAS~1\LOCALS~1\Temp\Rtmp2f270T\myfile.txt
      ⍕1↓∊(('\/'⍳r)=2+⎕io)⊂r),¨'\'
C:\DOCUME~1\AJAYAS~1\LOCALS~1\Temp\Rtmp2f270T\myfile.txt
```

1.5.2. tempfile()

This function returns the fully qualified name of a temporary file, suitable for R syntax.

```
> filename<-tempfile(,fileext=".log"); # Default location
> filename
[1] "C:\\DOCUME~1\\AJAYAS~1\\LOCALS~1\\Temp\\Rtmp2f270T\\filebcc3ad439e0.log"
```

Such a file can be used for, say, logging progress:

```
> connection<-file(filename,"w"); # option = r|w|a|rb|wb|ab i.e read|write|append [binary]
> writeLines("arbitrary text",connection);
> close(connection);
```

Until you close the handle or connection, it remains available for writing to the file from anywhere within your script.

1.6. Platform interface

The Windows version of R[4] provides a number of standard graphical dialogues. These dialogues are especially handy when you are running scripts that are time-consuming.

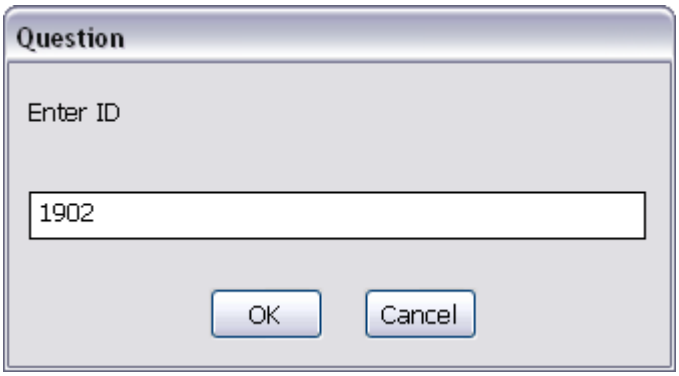
- Use the input box for introducing run time values into your script.
- Use the message box for managing pathways in your script depending on the options at runtime.
- Use the progress bar to provide visual feedback on progress.

1.6.1. Input box

This is similar to the Windows InputBox dialogue. This always returns a value: either the content of the edit box or NULL is returned depending on whether OK or Cancel is clicked.

Use one of the coercion functions to change the type of the returned value, e.g.

```
> pValue<-winDialogString("ID","0");
> as.numeric(pValue);
[1] 1902
```



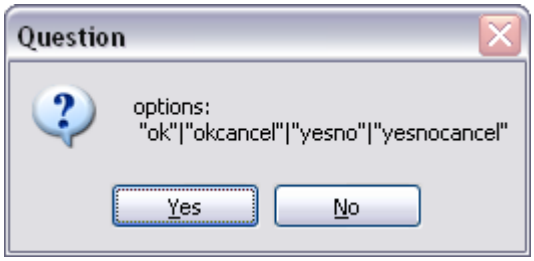
1.6.2. Message box

Use this dialogue to provide feedback and to prompt for interaction with the user.

```
> winDialog("yesno",'options:\r\n "ok"|"okcancel"|"yesno"|"yesnocancel"');
[1] "YES"
```

This dialogue returns the caption of the key that is clicked, in capitals.

Refer to the message shown in the picture: the options for the first argument. The choice of the first argument determines the caption of the dialogue.



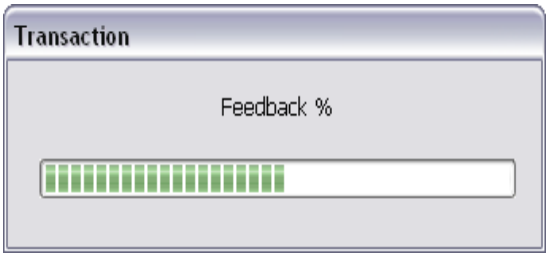
1.6.3. Progress bar

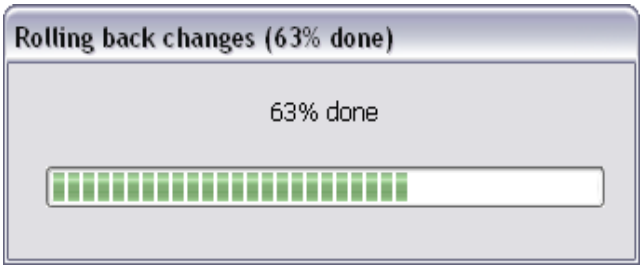
This dialogue provides visual verification of progress; it is useful for processes that take a very long time. The initial picture shows the start of the process and the second picture updates to indicate progress in terms of percentage.

Copy and paste the following code to re-create the demonstration.

```
pBar<-winProgressBar("Transaction", " Feedback %",0,100,50)
Sys.sleep(4)# Delay - change to a higher number if required
for(i in c(0, sort(runif(20, 0 ,100)), 100)) {
    Sys.sleep(0.1)
    info <- sprintf("%d%% done", round(i))
    setWinProgressBar(pBar, i, sprintf("Rolling back changes (%s)", info), info)
}
Sys.sleep(0.5); # Delay
close(pBar); # Close
```

[4] R does not have access to Win32 API calls.





The script closes the dialogue automatically at the end.

1.6.4. Shell

This function allows an operating system command to be executed. For full details of this command, see:

```
> ?shell
```

For example, in order to obtain a directory listing, the command is:

```
> a<-shell("dir",translate=TRUE,intern=TRUE);
```

Note the assignment of the results to an R variable: specifying intern=TRUE pipes the results into an R object in the R session.

1.6.5. System

This function allows an operating system command or platform application to be launched. For full details of this command, see:

```
> ?system
```

For example, in order to launch APL+Win, the command is:

```
> system('"C:/Program
Files/APLWINV11/aplw.exe"',minimized=FALSE,intern=FALSE,wait=FALSE,invisible=FALSE);
```

Note that the file specification needs to use / instead \ and that it is enclosed in both single and double quotes. R requires the command to be enclosed in quotes—the single quote—and Windows requires any path that has embedded spaces to be enclosed in quotes—the double quote.

This corresponds directly with the APL+Win □cmd system function.

1.7. Workspace/session management

By default, R and APL+Win both use workspaces, a binary dump of the active session; workspaces are archived to and re-created from files. There are similarities in the languages' management of their respective workspaces.

	APL+Win	R[5]
Default file extension	.W3	.RDATA
Default file location	This is the *Start In* property of the shortcut of each language.	
Drop workspace	)DROP filename[6]	unlink(filename)
Clear workspace	)CLEAR	rm(list=ls(all=TRUE)
Clear Session	Ctrl+A | Delete	Ctrl+L
Save workspace	)SAVE filename	save.image(filename) Or save(object1,object2,..object3,file=filename) save.image()
Load workspace	)LOAD filename	load(filename,[environment][7])

[5] R commands are case-sensitive.
[6] The file name must be fully qualified. Use double \ or single / to separate the path tree.
[7] The optional environment argument specifies a namespace which is the global one by default.

Active session profile

	APL+Win	R[5]
View contents of workspace	⎕nl 2 3 And '#' ⎕wi 'children'	ls.str() Or objects();
Copy all executable lines	CTRL+A | CTRL+Shift+G This copies the executable lines to the clipboard.	history(Inf) This opens a script editor session which contains all the lines executed in the current session.

Note:

- APL+Win and R workspaces are not interchangeable or usable by each other; this is very likely a blessing in disguise!
- APL+Win can copy a custom list of objects from a workspace file, that is from disk, but cannot save such a list to a workspace file. The reverse applies to R.
- Query and change the default file location using `⎕chdir ''` and `⎕chdir` *existingPath* with APL+Win and getwd() and setwd(*existingPath*) with R. When changing paths, they must already exist otherwise an error occurs. The Start In property of shortcuts promotes a sound mechanism for keeping projects in distinct locations—this implies that you should setup different shortcuts for each project.
- APL+Win workspaces may have a latent expression that is sometimes used to configure the session automatically by reading additional objects from other workspace, component files, or databases. R workspaces do not have latent expressions; R workspaces can be configured via script file(s) which may load workspaces cumulatively and fix functions and variables from the script file(s).
- Workspaces provide a very convenient mechanism for making ready the APL+Win and R sessions either for interactive use or for use as COM servers or COM clients.
- I am not going to use namespaces or code snippets that involve complex numbers: APL+Win has neither of them.

1.7.1. Active session profile

APL+Win has a rich set of primitive functions that query selective aspects of the active session. R has similar features.

```
> # Clean session
> rm(list=ls());
> #Create a numeric matrix
> nMatrix<-matrix(c(1,2.3,5,4),ncol=2);
> # Get a predefined data structure
> data(mtcars);
> #Create a character vector
> cVector<- strsplit("this is a sentence"," ");
> # Define a function
> AllObjects<-function(oopType="class"){
+   return(as.matrix((sapply(.GlobalEnv, FUN=oopType)),ncol=1))
+ }
> AllObjects();          # Will query "class"
           [,1]
AllObjects "function"
nMatrix    "matrix"
cVector    "list"
mtcars     "data.frame"
> AllObjects("typeof"); # query "typeof"
           [,1]
AllObjects "closure"
nMatrix    "double"
cVector    "list"
mtcars     "list"
> AllObjects("mode"); # query "mode"
           [,1]
AllObjects "function"
nMatrix    "numeric"
cVector    "list"
mtcars     "list"
> AllObjects("is"); # query list of classes
           [,1]
```

```
AllObjects Character,3
nMatrix    Character,4
cVector    Character,2
mtcars     Character,4
> #Result gives the type of elements and number of elements e.g.
> is(AllObjects);
[1] "function"         "OptionalFunction" "PossibleMethod"
> # Define another function
> ObjectSize<-function(){
+ return(as.matrix(sapply(sapply(ls(.GlobalEnv),get),object.size),ncol=1))
+ }
> ObjectSize(); # Size of active session objects in bytes
           [,1]
AllObjects 2892
cVector     208
mtcars     5336
nMatrix     144
ObjectSize 2884
> # Memory
> memory.size()  # reports the current or maximum memory allocation of the malloc function
used in this version of R.
[1] 12.88
> memory.limit() # reports or increases the limit in force on the total allocation.
[1] 1535
> # memory.limit(100) # allocates 100MB
```

All the facilities illustrated should be familiar to an APL+Win developer with one possible exception, namely, the reference to .GlobalEnv: this is the name of the default namespace. The R primitive ls() defaults in scope to its environment—in this case the function in which it is used—and it is necessary to specify the default namespace to obtain session rather than scope items or values.

1.8. R data structures

A basic understanding of R data structures is critical for two reasons.

- First, the data structures underpin the statistical and data visualisation facilities that R offers; therefore, some fluency with the data structures is a pre-requisite to exploring R.
- Second, it is necessary to understand how to manage data acquisition and transfer between R and APL+Win—in a COM client/server coupling context—to be able to draw on the R and APL+Win respective features.

The natural data structure in R is a vector--every other structure is a vector with appropriate attributes; for example, a matrix is a vector with a dimensions attribute. The key primitive function in defining any data structure is concatenation, for example, c(1,2,3) is a vector of three elements. Note that elements are always separated by comma, unlike APL+Win, which also allows space as the separator.

1.8.1. Built-in structures

A *matrix* is a subset of the *array* class; it always has two dimensions. A *vector* is also a subset of the *array* class; it always has one dimension. *Scalar*s do not exist; a scalar is simply a vector of length one. Each of these contain homogeneous data of type numeric, character, complex, or logical with or without missing values—designated as NA.

Structure	Dimensions	Type	Verification
Homogeneous			
vector	1	Numeric | Character | Complex | Logical	is.vector(*obj*| value)
array	n	Numeric | Character | Complex | Logical	is.array(*obj*| value)
matrix	2	Numeric | Character | Complex | Logical	is.matrix(*obj*| value)
factor	1	Numeric | Character	is.factor(*obj*| value)
table	2	Numeric	Is.table(obj)
Heterogeneous[8]			

[8] These data structures have a pre-defined format; in other words, they are not free-form as APL+Win nested variables.

Data type coercion

data.frame	Numeric, Character, Complex, Logical	is.data.frame(*obj*)
ts	Numeric, Character, Complex, Logical	Is.ts(*obj*)
list	Numeric, Character, Complex, Logical, Function, Expression, Formula	is.list(*obj*)

- APL+Win does not support the complex data type—except in user-defined code—so it will remain out of scope of this article.
- R missing values, NA, do not have an equivalent in APL+Win; this will also remain out of scope.

1.8.2. Verifying data type

Unlike strongly typed languages such as C#, R, like APL+Win, does not permit the type of variables to be declared prior to assignment of values. Variables come into existence upon assignment and their type is inferred within context. Also, like APL+Win variables, R permits its objects to be re-assigned data of the same or different type.

It is interesting to note that there is no 'is.Date' function—you need to use install another package for this or write your own function.

The 'as.Date' function takes a number (days elapsed from the origin) as its first argument and a date origin as the second.

```
> as.Date(23.2333,origin="2000-01-01") # Ignores the fractional part
[1] "2000-01-24"
> as.Date(23.2333,origin="1970-01-01") # Ignores the fractional part
[1] "1970-01-24"
```

For intensive date arithmetic, it might be better to use APL+Win than R, if only because with API calls, you can cope with not only locale but also time, that is, with time stamps.

The following table summarises the common method for querying the type of an object.

Type	**Syntax**	**Result**
Numeric	is.numeric(obj | value)	TRUE | FALSE
Character	is.character(obj | value)	TRUE | FALSE
Complex	is.complex(obj | value)	TRUE | FALSE
Logical	is.logical(obj | value)	TRUE | FALSE
Function	is.function(obj)	TRUE | FALSE
Expression	is.expression(obj | value)	TRUE | FALSE
Formula	-	-

1.8.3. Data type coercion

Where objects (that is variables) are conformable, R has built-in functions for converting one type of data to another; failure raises an error and if the result is assigned to a variable, the value is NA[9].

- as.numeric (obj | value) coerces its argument to numbers.
- as.character(obj|value) coerces its argument to character.
- as.complex(obj|value) coerces its argument to complex.
- as.logical(obj|value) coerces its argument to logical; the two states are TRUE and FALSE.

2. Using R as a Server to an APL+Win Client

I have found three ways of deploying R as a server to an APL+Win client.

1. Rserve – a TCP/IP server. APL+Win can use `⎕ni`; however, it is not that simple.
2. R as a background server.
3. R as a foreground server.

[9] The value NA corresponds to NULL values: a NULL value is not equal to any other value, including another NULL value.

2.1. What is Rserve?

The Rserve homepage, http://www.rforge.net/Rserve/index.html, answers this question as follows: "**Rserve** is a TCP/IP server which allows other programs to use facilities of **R** (see www.r-project.org) from various languages without the need to initialize **R** or link against **R** library. Every connection has a separate workspace and working directory. Client-side implementations are available for popular languages such as C/C++, PHP and Java. **Rserve** supports remote connection, authentication and file transfer. Typical use is to integrate **R** backend for computation of statistical models, plots etc. in other applications. "

A pre-requisite to using Rserve is a client; see http://www.rforge.net/Rserve/dev.html for the specification. A client implemented in C# is at http://sourceforge.net/projects/rservelink/files/. For other documentation, see http://www.rforge.net/Rserve/.

This offers another means of APL+Win and R client/server coupling. There are several options, including writing a client in APL+Win using `⎕ni`, writing a COM interface using C# that will act between Rserve and APL+Win etc.

At this stage, this is still very much research in progress; it offers another option for using R as a Server with an APL+Win client.

2.2. R Server: Foreground or Background instance?

R can act either as an in-process server, that is, in the background and invisible or as an out-of-process server, that is, in the foreground and visible.

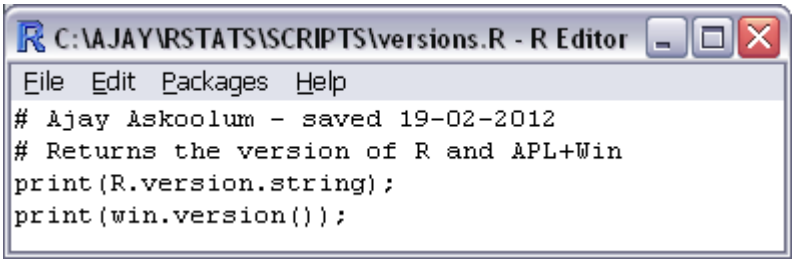
In deployment, the preferred mode must be to use R in the background as this promotes a more robust arrangement—nothing can make changes to the session except the client and this mode may be the faster of the two.

During application development, either mode will work fine but a foreground server may be more desirable.

- Interaction with the R server is possible: you can examine and make changes in its session.
- The full functionality of the R server is available from its console: you can switch to it and execute commands or invoke help on R primitives etc.
- Initially, code construction and debugging is much easier from the server's console than from APL+Win's interactive mode simply because you eliminate the overhead of translating R code into APL+Win code.

My own preference is to use the background server with debugged scripts and the foreground server to construct the scripts. The R expression history(Inf) captures all lines executed in the current session, including itself, into an unnamed script file that is shown in a window.

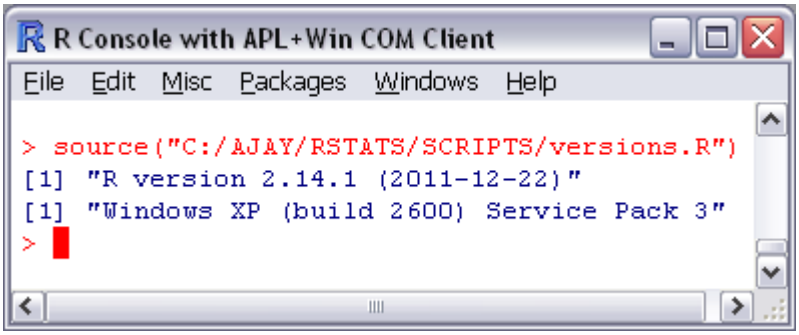
The script may be edited as required; usually, you should add a comment relating to the creation date and purpose of the file.



The script may be re-executed with the following expression:

source("C:/AJAY/RSTATS/SCRIPTS/versions.R")

From the user interface, use **File | Source R Code …**



- Save a script with extension R using **File | Save**; note that the file is a plain text file.
- One script file may execute other script files using their respective fully qualified name; however, history(Inf) will show the reference to these other files rather than their respective contents.

2.2.1. R as a foreground server

In order to use R as an out-of-process or foreground server, R must be running. It is convenient to launch R from APL+Win using the following function.

```
    ∇ RFServer;⎕elx
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕elx←'→⎕lc'
[3]   :if 0=⍬⍴⎕wcall ¨(⊂'FindWindow'),¨0,¨⊂¨'R Console' 'R Console with APL+Win
      COM Client'
[4]       ⍝ Start R & wait until an R session has started.
[5]       0 0⍴⎕WCALL 'WinExec' 'C:\Program Files\R\R-2.14.2\bin\i386\Rgui.exe' 'SW
      _SHOWMINIMIZED'
[6]   :endif
[7]   ⎕wself←'⍋rfs' ⎕wi 'Create' 'RCOMServerLib.StatConnector'
[8]   ⎕wi 'XInit' 'R'
[9]   ⎕wi 'XEvaluateNoReturn' 'utils::setWindowTitle("with APL+Win COM Client")'
[10]  ⍝ '⍋rfs' ⎕wi 'Create' '{3660C348-DF59-4CA2-83E8-3A913A9FBC77}' ⍝ Using clsid
[11]  0 0⍴⎕wcall 'ShowWindow' ('#' ⎕wi 'hwndmain') 'SW_SHOWNORMAL'
[12]  '#' ⎕wi 'caption' "APL+Win using R as COM Server"
    ∇
```

On line [5], specify the fully qualified name of the R executable; this is easily found in the target box of the R desktop shortcut.

I am using the standard R graphical user interface (GUI), R Console: if you opt for another interface, the name of the executable will differ.

Starting the R server may take a variable length of time with the consequence that line [7] may occasionally fail. In order to circumvent this, line [2] establishes an error handler that causes line [7] to be re-executed. Be warned that if you do not have a working version of R, this function will execute indefinitely.

In order to provide a means of visually identifying the server and client, both their captions are changed in lines [9] and [12] respectively.

2.2.2. R as background server

Creating an in-process or background instance of R server does not affect any running R sessions. The following function creates an in-process instance of R as a server.

```
    ∇ RBServer
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕wself←'⍋rbs' ⎕wi 'Create' 'StatConnectorSrv.StatConnector'
[3]   ⎕wi 'XInit' 'R'
[4]   ⎕wi 'XEvaluateNoReturn' 'utils::setWindowTitle("R with APL+Win COM Client")'
[5]   ⍝ Or '⍋rbs' ⎕wi 'Create' "{18C8B662-81A2-11D3-9254-00E09812F727}" ⍝ using clsid
[6]   '#' ⎕wi 'caption' "APL+Win using R as COM Server"
    ∇
```

Note that the background instance can be created regardless of whether there is a visible R session; a new instance is created always.

2.2.3. R Server considerations

Irrespective of whether you choose to use an in-process or an out-of-process server, it is advisable to close all existing R sessions before creating the server instance. This eliminates the risk of confusion—you know exactly which instance of R is acting as the server—and ensures that machine resources are available for the R server session.

As a further visual verification, lines [9] and [4] of RFServer and RBServer, respectively, modify the caption of the R session, although the instance created by RBServer is never visible.

Neither type of server exposes a 'Visible' property so the decision as to which type of instance is preferable is a foregone conclusion as soon as you decide whether you want a visible or invisible session. Both instances expose the same set of methods and neither exposes any properties or events.

2.2.4. Managing R objects from APL+Win

If you need user-defined functions or variables in R, the most expedient approach is to define and debug the functions in R and then save the definition either as a script or you can save the R workspace. APL+Win can then run the script or load the R workspace programmatically. This ensures that the R instance contains bug free code and variables.

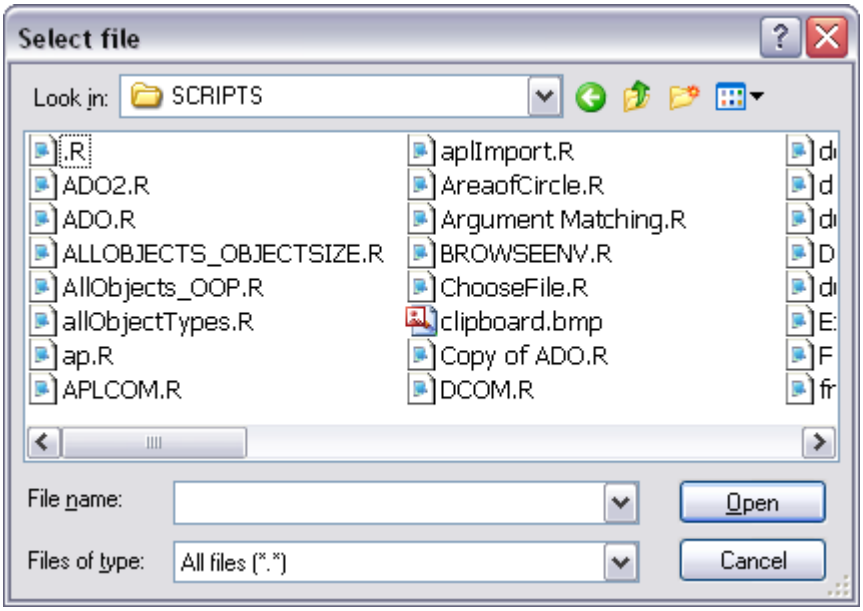
```
# Saving the R Workspace
setwd("c:/temp")          # Set the default location
save.image("file.rdata") # Extension is RDATA. Save the workspace
load("file.rdata")        # Load the workspace from the default location
save.image("c:/temp/file2.rdata) # Saving with fully qualified name
load("c:/temp/file2.rdata)       # Loading with fully qualified name
```

Saving the workspace saves all the contents in the active session. A workspace can be saved or loaded from within APL+Win.

```
      ⎕wi 'XEvaluateNoReturn' 'save.image("c:/temp/ajay.rdata")' ⍝ Save the active WS
      ⎕wi 'XEvaluateNoReturn' 'load("c:/temp/ajay.rdata")'       ⍝ Load an R WS
```

In order to select a workspace or file under programme control, the following command invokes a graphical interface for file selection:

```
> file.choose();
```



The dialogue returns the name of the file; if it is a workspace, it may be opened by the following APL+Win function.

```
.    ∇ LoadRWS;⎕elx;file
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ Choose a workspace using a GUI and load it.
[3]   ⎕elx←'→⎕lc+1'
[4]   file←⍬
[5]   file←⎕wi 'XEvaluate' 'file.choose()'
[6]   :if 0≠⍴file
[7]       ⎕wi 'XEvaluateNoReturn' ('load(#)' InLineExpression file)
[8]   :endif
      ∇
```

In order to load a script—this can contain multiple variables and functions—into the active R session, use the following expression.

```
      ⎕wi 'XEvaluateNoReturn'  'source("C:\\AJAY\\RSTATS\\SCRIPTS\\AreaofCircle.R")'
      ⎕wi 'XEvaluateNoReturn'  ('source("")' InLineExpression file)
```

The argument *file* specifies either a file name retrieved from the *file.choose()* dialogue or is hardcoded.

The configuration of the R server session using either a workspace or a script file is faster and less error prone than achieving the same task from APL+Win.

2.2.5. R Functions

It is possible to create functions and variables programmatically from the APL+Win client; it is imperative that you use a fail-safe naming convention for your identifiers as the penalty may be a locked APL+Win session. The APL+Win naming convention—without any APL characters—is fail-safe within R.

R functions are a simple string vector: an APL+Win string of rank 1. Edit and define function definitions as follows:

```
      fnDef←' '
      )edit fnDef
      fnDef ⍝ Examine the contents
aoc<-function (radius = 0)
{
    radius[is.na(radius)] <- 0;
    return(pi * radius^2);
}

      ⍴fnDef
92
     ⎕wi 'XEvaluateNoReturn' (fnDef~⎕tcnl)
      ⍝ Call the function
      ⎕wi 'XEvaluate' 'aoc(6)'
113.0973355
      ⎕wi 'XEvaluate' 'aoc()' ⍝ Default value will be used
0
```

- Functions need to be fairly robust; this function defaults its single argument to 0, and treats missing values as 0.
- R functions naturally extend to take any suitable R data structure as arguments. In this case, the function is defined with a scalar argument. However, it will cope with arguments of other data structures.

```
      ⎕wi 'XEvaluate' 'aoc(c(2,NA,4.5))'
12.56637061 0 63.61725124
```

2.2.5.1. Passing arguments from APL+Win

An alternative to creating the argument in R or embedding the arguments is to use the following syntax:

```
      ⎕wi 'XEvaluate' ('aoc(c())' InLineExpression  'c(9,10,10)')
254.4690049 314.1592654 314.1592654
```

The function InLineExpression combines the function name and its arguments and returns it as a string; this avoids having to create a variable in R to hold the arguments and the function call is completed in a single step.

It is a little cumbersome to embed the arguments of a function: the XEvaluate method takes only one argument. A suitable workaround is to define the argument(s) of any function as variables in the R session and then to call the R function.

```
      ⎕wi 'XSetSymbol' 'rad' (2 3⍴2 4 0 12 1.5 5)
      ⎕wi 'XEvaluate' 'aoc(rad)'
  12.56637061 50.26548246    0
 452.3893421   7.068583471 78.53981634
```

Although it is much more convenient to pass an APL+Win variable as the value of an R variable, sometimes it is expedient to code the values manually; for instance, if you want to include an R value type line NA.

```
      ⎕wi 'XEvaluateNoReturn' 'rad2<-c(2,4,NA,12,1.5,5);'
      ⎕wi 'XEvaluate' 'aoc(rad2)'
12.56637061 50.26548246 0 452.3893421 7.068583471 78.53981634
```

2.2.5.2. Functions are objects

R functions are objects; therefore, they pass their attributes to another object when assigned.

```
      ⎕wi 'XEvaluateNoReturn' 'f<-get("aoc",mode="function");'
      ⎕wi 'XEvaluate' 'f(c(1:5));'
3.141592654 12.56637061 28.27433388 50.26548246 78.53981634
```

It is also possible to execute the function using the following syntax:

```
      ⎕wi 'XEvaluate' 'do.call("aoc",list(c(1:5)));'
3.141592654 12.56637061 28.27433388 50.26548246 78.53981634
```

Note that when executed this way, the arguments must be specified as a *list*.

2.2.5.3. Anonymous functions or lambda expressions

R permits function definitions without name, that is, lambda expressions. Such functions are defined and executed all at once. An example:

```
> AreaOfRectangle<-function(x,y){
+ z<-(function(x,y){return(x*y);})(x,y); # Lambda Expression
+ return(z);
+ }
> x<-sample(1:10,4); # 4 random numbers between 1 and 10
> y<-sample(1:10,8); # 8 random numbers between 1 and 10
> y
[1]  3  6  5 10  2  8  1  4
> y
[1]  3  6  5 10  2  8  1  4
> AreaOfRectangle(x,y);
[1]  9 42 10 60  6 56  2 24
```

2.2.5.4. Vagaries of R Scoping Rules

R has the concept of environment. In an R session, the environment is .GlobalEnv. and can be seen as a namespace. However, R user-defined functions create their own environment or scope. This can cause some nasty surprises simply because the source code—where the variable is declared—may fail to provide any clues about the scope of variable.

R uses lexical scoping: that means that if an identifier is referenced within the scope of a function (between double braces), the value of that identifier is sought within that local scope and if not found it will be sought in the environment scope. If the environment does not have that identifier, an error occurs. The scope of a calling function is never examined to determine if that identifier existed therein. The following example illustrates this point.

```
> envSV<-100; # This identifier has environment scope
> Func1<-function(){
+ envSV<-200.175; # This has function scope ... it will never be used by func2()
+ return(Func2(20.78));
+ }
> Func2<-function(arg){
+ return(arg+envSV);
+ }
> # call func1 which in turn calls func2
> Func1()
[1] 120.78
> # Environment Value(100) plus 20.78
> # Erase the Environment Value ... func1() should fail
> rm(envSV); # or remove(envSV)
> Func1()
Error in func2(20.78) : object 'envSV' not found
```

APL+Win uses Dynamic Scoping.

- A calling function passes all it local variables to the called function.
- Any variable localised in the calling function masks that variable: a variable of the same name in the global scope is inaccessible.

The corresponding APL+Win code and difference in behaviour is illustrated below.

```
      envSV←200
      ⎕vr ¨ 'Func1' 'Func2'
     ∇ Z←func1;envSV
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   envSV←200
[3]   Z←Func2 20.78
    ∇
     ∇ Z←Func2 R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   Z←R+envSV
    ∇

      Func1
220.78
      )erase envSV
      Func1
220.78
```

```
      ⍝ Func2 used envSV that is local to its calling function Func2 AND not its global
instance
```

2.2.5.4.1. Closure

Like many languages, R has the concept of 'Closure'; this does not exist in APL+Win. Essentially, this has a critical bearing on understanding the scope of R objects. Refer to these links for an explanation of what this means.

http://www.lemnica.com/esotericR/Introducing-Closures/	R Language
http://www.codethinked.com/c-closures-explained	C# Language

2.2.5.5. Vagaries of R functions and argument matching

APL+Win functions are either niladic (no argument), monadic (one arguments, always as the right-hand argument), dyadic (two arguments, one left and one right- hand arguments), or ambivalent (as dyadic, except that the left-hand argument is optional). Moreover, by default, the conventional APL+Sin function signature does not permit argument matching by position.

All R functions are ambivalent and polyadic irrespective of the way they are coded, their arguments can have default values, and arguments can be passed by name or by position.

```
> # No error trapping
> RFuncVer1<-function(R,L){
+ return(L+R);
+ }
> RFuncVer1();
Error in L + R : 'L' is missing
> #Incomplete error trapping
> RFuncVer2<-function(R,L){
+   if (missing(L)){
+      L<-10;
+      }
+ return(L+R);
+ }
> RFuncVer2();
Error in L + R : 'R' is missing
> # Better signature verification
> RFuncVer3<-function(R,L){
+   if (missing(L)){
+      L<-10;
+      }
+   if (missing(R)){
+      R<-10;
+      }
+ return(L+R);
+ }
> RFuncVer3();
[1] 20
> # Much simpler approach to cope with ambivalence
> RFuncVerAlt<-function(R=10,L=10){
+ return(L+R);
+ }
> RFuncVerAlt();
[1] 20
> # NOTE: There is still no verification for the type of L and R
> RFuncVerAlt(,9); # Argument by position
[1] 19
> RFuncVerAlt(1,9); # Argument by position
[1] 10
> RFuncVerAlt(R=9);# Argument by Name
[1] 19
```

R has a further subtlety in matching arguments. Consider the following function.

```
> ArgMatch<-function(sideFirst=1,sideFinal=10){
+ return(sideFirst+sideFinal);
+ }
> ArgMatch(sideFi=10,5);  # Does not resolve uniquely
Error in ArgMatch(sideFi = 10, 5) :
  argument 1 matches multiple formal arguments
> ArgMatch(sideFin=10,5); # By PARTIAL name & position
[1] 15
```

The foregoing example illustrates the following:

- Arguments can be passed using name and position simultaneously; this may re-order the sequence in which the arguments are specified.
- Arguments by name can be passed by specifying an abbreviation of the argument name that resolves uniquely.
- Remember to use = and not <- to pass arguments by name; in other contexts, = and <- are interchangeable.

2.2.6. R Variables

Experience of APL+Win COM automation teaches two valuable lessons.

First, when the server expects values of type decimal *and* it cannot coerce passed-in values to decimal seamlessly, any attempt to pass such values from APL+Win fails as APL+Win cannot pass such values outright. You would need to workaround this problem. However, R, like APL+Win is a type-inferred language. Therefore, this is not an issue.

Second, when the server accepts arrays, APL+Win is well-placed for sending and receiving such values subject to two overriding issues.

- The shape of simple string array variables in APL+Win returns the number of rows and columns and the number of columns is always uniform; APL+Win pads all strings with spaces, where necessary. Other languages return the number of columns as 1, irrespective of the number of characters on each row and they do not pad the rows with spaces. Therefore, whenever such arrays are transferred or received, APL+Win needs to transfer the array as a nested variable (without padding) and disclose them on receipt.
- APL+Win is row-major; if the target language is not also row-major, the acquisition and transfer process needs to transpose the array either at the client or the server. Unlike APL+Win, R is column-major[10].

Note that the term ‘array’ is used here in the APL+Win sense; in R jargon, an APL+Win array of rank 2 is called a ‘matrix’ and those of higher rank are called ‘array’.

2.2.6.1. String Arrays

This is a simple illustration of the two issues with string arrays. The following creates the variables in R.

```
> # String Arrays in R
> weekDaysVector<-c('Monday','Tue','Wed','Thu','Fri','Sat','Sun');
> weekDaysVector;
[1] "Monday" "Tue"    "Wed"    "Thu"    "Fri"    "Sat"    "Sun"
> weekDaysMatrix<-matrix(c('Monday','Tue','Wed','Thu','Fri','Sat','Sun'),ncol=1);
> weekDaysMatrix;
     [,1]
[1,] "Monday"
[2,] "Tue"
[3,] "Wed"
[4,] "Thu"
[5,] "Fri"
[6,] "Sat"
[7,] "Sun"
> dim(weekDaysMatrix);
[1] 7 1
```

Reading the same variables in APL+Win:

```
      GetStringArrays
 ... first Vector
Getting weekDaysVector
 Monday Tue Wed Thu Fri Sat Sun
Shape and Rank
 7  2
Disclose and shape after Disclose
 Monday  7 6
 Tue
```

```
    ∇ GetStringArrays
[1]   ⍝ Ajay Askoolum - APL2000 Conference
April 22-24, 2012
[2]   ' ... first Vector'
[3]   'Getting weekDaysVector'
[4]   ⎕wi 'XGetSymbol' 'weekDaysVector'
[5]   'Shape and Rank'
[6]   (⍴⎕wi 'XGetSymbol' 'weekDaysVector')
(≡⎕wi 'XGetSymbol' 'weekDaysVector')
```

[10] Excel is another column-major environment.

```
 Wed
 Thu
 Fri
 Sat
 Sun
Transpose has no effect on vectors
 ... next Matrix
 Monday Tue Wed Thu Fri Sat Sun
Shape and Rank
 1 7  2
Did you notice the transposition?

Need to 1. Ravel 2. Transpose 3. Disclose
the matrix
7 6
Did you notice the padding?
 Check how it looks
Monday
Tue
Wed
Thu
Fri
Sat
Sun
```

```
[7]   'Disclose and shape after Disclose'
[8]   (⊃⎕wi 'XGetSymbol' 'weekDaysVector')
(⍴⊃⎕wi 'XGetSymbol' 'weekDaysVector')
[9]   'Transpose has no effect on vectors'
[10]  ' ... next Matrix'
[11]  ⎕wi 'XGetSymbol' 'weekDaysMatrix'
[12]  'Shape and Rank'
[13]  (⍴⎕wi 'XGetSymbol' 'weekDaysMatrix')
(≠⎕wi 'XGetSymbol' 'weekDaysMatrix')
[14]  'Did you notice the transposition?'
[15]  ' '
[16]  'Need to 1. Ravel 2. Transpose 3.
Disclose the matrix'
[17]  (⍴⊃,⎕wi 'XGetSymbol'
'weekDaysMatrix')
[18]  'Did you notice the padding?'
[19]  ' Check how it looks'
[20]  ⊃,⎕wi 'XGetSymbol' 'weekDaysMatrix'
      ∇
```

Without Ravel, the resulting matrix will be three-dimensional:

```
      ⍴⊃⍪⎕wi 'XGetSymbol' 'weekDaysMatrix'
7 1 6
      ⍴⍪⊃⎕wi 'XGetSymbol' 'weekDaysMatrix'
6 7 1
```

The most convenient—that is, with least overhead—approach to dealing with string vectors or matrices acquired from another environment such as R is to keep them as nested vectors of depth 2.

```
      ⎕wi 'XGetSymbol' 'weekDaysVector' ⍝ As nested vector ... Rank 1,depth 2
 Monday Tue Wed Thu Fri Sat Sun
      ⍴¨⎕wi 'XGetSymbol' 'weekDaysVector' ⍝ ... and no padding
 6  3  3  3  3  3  3
      ,[⍬]⎕wi 'XGetSymbol' 'weekDaysVector' ⍝ As nested matrix ... Rank 2, depth 2
 Monday
 Tue
 Wed
 Thu
 Fri
 Sat
 Sun
      ∈⍴¨,[⍬]⎕wi 'XGetSymbol' 'weekDaysVector' ⍝ ... and no padding
6 3 3 3 3 3 3
```

2.2.6.2. Array collation sequence

For clarification:

- Row-major means that values are used to populate rows first and then columns.
- Column-major means that values are used to populate columns first and then rows. R is column-major by default but it does have the option to construct row-major matrices.
- Row- or column- major is not indicative of how the matrices are actually stored in memory.

First, construct arrays in R; the dimensions are readable from the results.

```
> matA<-matrix(c(1,2,3,4,5,6),nrow=2);
> matB<-matrix(c(1,2,3,4,5,6),ncol=3);
> matA;
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> matB;
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> identical(matA,matB);
[1] TRUE
> matC<-matrix(c(1,2,3,4,5,6),ncol=3,byrow=TRUE); # Note collation sequence
> matC;
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Next, the same results in APL+Win:

```
      matA←2 3⍴1 2 3 4 5 6
      matC←⍉2 3⍴1 2 3 4 5 6
      matA matC
  1 2 3     1 4
  4 5 6     2 5
            3 6
```

The impact of the difference in collation sequence is apparent when the R variables are retrieved within APL+Win.

```
      matA≡⍴⎕wi 'XGetSymbol' 'matA' ⍝ Transposition within APL+Win
0
      matA≡⍴⎕wi 'XEvaluate' 't(matA)' ⍝ Transposition within R
0
      (⍴matA) (≡matA) (⍴⍉matA) (≡⍴⍉⎕wi 'XGetSymbol' 'matA')
 2 3   1   3 2   1
```

So, do the matrices created in APL+Win and retrieved from R match? The answer is in the implicit coercions that take place transparently.

```
      ⎕wi 'XEvaluate' 'typeof(matA)'
double
      ⎕dr matA ⍝ 323 is integer, 645 is double
323
```

In contrast, when an R variable is created by APL+Win, there is confusion with neither the dimension nor the type:

```
      ⎕wi 'XSetSymbol' 'matAAlt' matA
      matA≡⎕wi 'XGetSymbol' 'matAAlt' ⍝ Note that there is no explicit transposition
1
```

The problems arise because of the co-existence of variables created in R from within and without, that is, from APL+Win.

2.2.6.2.1. What transposition?

In the latter example, the COM interface transposes the APL variable before sending it to R and on receiving it back from R thereby cancelling the effect. Is it safe to conclude that when APL+Win creates a variable and then retrieves its value, there is no confusion relating to collation sequence?

```
      (⎕wi 'XEvaluate' 'rowSums(matAAlt)') (⎕wi 'XEvaluate' 'colSums(matAAlt)')
 5 7 9   6 15
      (+⌿matA) (+⌿⍉matA) ⍝ Remember matAAlt is matA
 6 15   5 7 9
```

Note that the sum of APL+Win rows appears to match the sum of columns returned by R and vice-versa. Unless I am confusing matters, it appears that the collation sequence has an impact even when APL+Win creates the variables and needs clarification. A simple investigation is to create two conformable matrices using APL+Win and to calculate and compare their product from each language.

```
      MatMult
Do A in APL+Win and R thet
Match?
1
Do B in APL+Win and R thet
Match?
1
 Acid Test Calculate AB
(matrix multiply)
 First in APL+Win
 273 455
 243 235
 244 205
 102 160
 Next in R
⎕WI ERROR: exception 8004000B
MatMult[17] ⎕wi 'XEvaluate' 'A
%⍟% B'
            ^
```

```
    ∇ MatMult
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24,
2012
[2]   A←4 3⍴14 9 3 2 11 15 0 12 17 5 2 3
[3]   B←3 2⍴12 25 9 10 8 5
[4]   ⎕wi 'XSetSymbol' 'A' A
[5]   ⎕wi 'XSetSymbol' 'B' B
[6]   'Do A in APL+Win and R thet Match?'
[7]   A≡⎕wi 'XGetSymbol' 'A'
[8]   'Do B in APL+Win and R thet Match?'
[9]   B≡⎕wi 'XGetSymbol' 'B'
[10]  ' Acid Test Calculate AB (matrix multiply)'
[11]  ⍝ Remember:
[12]  ⍝ the number of rows of 1st matrix must match
number of columns of 2nd
[13]  ⍝ the dimensions of the product matrix is Rows of
1st matrix × Columns of 2nd
[14]  ' First in APL+Win'
[15]  A+.×B
[16]  ' Next in R'
[17]  ⎕wi 'XEvaluate' 'A %⍟% B'
    ∇
```

Why the error? The obvious culprit is that the matrices are not conformable for multiplication—this is confirmed with R.

```
>A;
     [,1] [,2] [,3] [,4]
[1,]   14    2    0    5
[2,]    9   11   12    2
[3,]    3   15   17    3
> dim(A);
[1] 3 4
> B;
     [,1] [,2] [,3]
[1,]   12    9    8
[2,]   25   10    5
> dim(B);
[1] 2 3
```

Within R, the variables are transposed! Therefore the resolution must be to transpose the matrices when calculations based on them are carried out within R.

```
> t(A) %*% t(B);
     [,1] [,2]
[1,]  273  455
[2,]  243  235
[3,]  244  205
[4,]  102  160
```

```
      A+.×B
273 455
243 235
244 205
102 160
```

Therefore, the product of A and B calculated in R and APL+Win must surely match?

```
      (⍎⎕wi 'XEvaluate' 't(A) %*% t(B)') - A+.×B ⍝ Either transpose the result in APL+Win
1
      (⎕wi 'XEvaluate' 't(t(A) %*% t(B))') = A+.×B ⍝ Or transpose the result in R
1
```

The inevitable conclusion is that it is necessary to transpose matrices when transferring or acquiring them from R using APL+Win—thereby enabling calculations within R without surprises—albeit transposition is unnecessary—it simply cancels out—when simply sending and receiving matrices to and from R.

2.2.6.2.2. Assigning variables

In general, the following syntax can be used for creating/reading variables.

	Assigning variables	Reading Variables

Option 1	`⎕wi 'XEvaluateNoReturn' 'var<-90'`
	`⎕wi 'XEvaluate' 'var'`
	`90`
		`⍝ Simply read`
		`⎕wi 'XGetSymbol' 'myVar'`

Option 2	`⎕wi 'XSetSymbol' 'var' 80`
	`10×⎕wi 'XGetSymbol' 'var'`
	`800`
		`⍝ Combined read/calculate`
		`⎕wi 'XEvaluate' 't(myVar);'`
		
		`⍝ Indirect read/calculate`
		`⎕wi 'XSetSymbol' 'indRead' (⎕ts)`
		`⎕wi 'XSetSymbol' 'varName' 'indRead'`
		`⎕wi 'XEvaluate' 'rev(get(varName));'`
		`2012 3 18 8 7 25 515`

Of these two options, the second is recommended as it allows better control: you can pass APL+Win workspace variables and results of expressions with or without coercion. Another benefit for adopting the second option is that you can add attributes to your variables—this is not possible in a one-pass operation. For one-pass assignments, you need to collate the R expressions and evaluate them in the R environment or save all the expressions in a script file and call R to execute the script.

```
⎕wi 'XEvaluateNoReturn' 'myMat<-matrix(c(1,2,3,4),ncol=2);' ⍝ Messy!
⎕wi 'XSetSymbol' 'myMat' (2 2⍴1 2 3 4)                     ⍝ Clean!
```

Another aspect of R raises another consideration, namely, the fact that R does not have scalars: R scalars are vectors of length 1. *In general*, R's list data structure and APL+Win's nested vectors (of depth 2) and nested arrays (of depth 2 and rank 2) are *roughly* compatible, given that with APL+Win, a scalar of length 1 and a scalar of length ⍬ are conformable. However, the depth of the APL+Win nested vector must be 2: vectors of higher depth raise an error in R—one more reason for creating variables from scripts and directly in R.

```
a←'APL+Win' 'nested' 'vectors/arrays' 'and' 'R' 'lists' 'are' 'comparable'
⎕wi 'XSetSymbol' 'aplNV' a
⎕wi 'XSetSymbol' 'aplNA' (2 4⍴a)
⍝ Remember transpose gets cancelled
```

```
      aplNV←⎕wi 'XGetSymbol' 'aplNV'
      aplNA←⎕wi 'XGetSymbol' 'aplNA'
      ∧/¨a=aplNV
1 1 1 1 1 1 1
      ⍝ But, the scalar issue comes into play!
      a-aplNV
0
      (↑¨⍴¨a) (↑¨⍴¨aplNV)
 7 6 14 3 0 5 3 10  7 6 14 3 1 5 3 10
```

APL+Win nested variables of depth greater than 2 *cannot* be passed directly into R.

```
      a←'This' 'happens' 100 ⋄ b←'percent' 'of' 'the' 'time'
      ⎕wi 'XSetSymbol' 'aplN' (a b)
⎕WI ERROR: 80010105 The server threw an exception.
      ⎕wi 'XSetSymbol' 'aplN' (a b)
      ∧
      -a b
3
      ⍝ Workaround
      ⎕wi 'XSetSymbol' 'tmpAPL1' a
      ⎕wi 'XSetSymbol' 'tmpAPL2' b
      ⎕wi 'XEvaluateNoReturn' 'aplN<-list(c(tmpAPL1,tmpAPL2))'
      ⎕wi 'XEvaluateNoReturn' 'rm(tmpAPL1,tmpAPL2)'
    ⍝ If I try to recover aplN from R, APL+Win freezes!
      ⎕wi 'XEvaluate' 'typeof(aplN)'
list
```

The reason APL+Win freezes is because we are now in the realm of R data structures (in this case, aplN is a list) and the package that sits between R and APL+Win was designed not for APL+Win but for Excel. Data structures require special care since an R data structure is an object that holds (and displays) values and attributes together. With APL+Win, the values and attributes must be retrieved separately.

2.2.6.2.3. Indirect reference

Variables may be referred to indirectly; consider this example:

```
      ⎕wi 'XSetSymbol' 'abc' (89 78 67)
      ⎕wi 'XSetSymbol' 'xyz' 'abc'
      ⎕wi 'XEvaluate' 'get(xyz);'
89 78 67
```

In this instance, the variable *xyz,* which holds the string *abc*, is coerced into returning the value of the variable *abc*. Calculations are possible on indirectly referenced variables.

```
      ⎕wi 'XEvaluate' 'sum(get(xyz));'
234
```

2.2.6.2.4. Assigned by value

Like functions, R variables are also objects. However, variables are assigned by value always and not by reference. In the example below, the variable *abcNew* inherits the value of the variable *abc*; and then it is changed. Whilst the new variable inherits the attributes—in this case it is its comment—of the original variable, the two variables remain distinct.

```
      ⎕wi 'XEvaluateNoReturn' 'abcNew<-abc;'
      ⎕wi 'XEvaluateNoReturn' 'abcNew[0 == (abcNew %% 2)]<--1;'
      ⎕wi¨ (⊂'XGetSymbol'),¨⊂¨'abc' 'abcNew'
 89 78 67  89 ¯1 67
```

2.2.7. R Objects: Functions and variables attributes

R objects are endowed with attributes; R combines its objects' value and attributes when it displays them in its own session. However, when APL+Win sends or retrieves an object to the R session, R only passes the values of the objects by default. And, APL+Win variables do not have some of the explicit or user-assigned attributes that R supports.

Therefore, APL+Win must pass the values and then the attributes whenever it creates an object in R. Equally, APL+Win must retrieve the values and then the attributes. Consider the following example, it illustrates the implications.

I have created a matrix of shape 2 3 in R using the following expression:

```
> # This session of R is the foreground server instance I am using from APL+Win
> # The objects created in this session are available from APL+Win
```

```
> myMatrix<-
matrix(c(1,2,3,4,5,6),dimnames=list(c('FirstRow','SecondRow'),c('Col1','Col2','Col3')),nrow=2,
ncol=3);
```

Its class is:

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

In R it displays as follows:

```
> myMatrix
          Col1 Col2 Col3
FirstRow     1    3    5
SecondRow    2    4    6
```

In APL+Win, only the values are transmitted[11]:

```
      ⎕wi 'XGetSymbol' 'myMatrix'
 1 2
 3 4
 5 6
```

The attributes must be queried separately.

```
      ⎕wi 'XEvaluate' 'dimnames(myMatrix)' ⍝ Fails to enumerate
 FirstRow Col1
```

Alternative methods for recovering the attributes:

```
      ]display (⊃⎕wi 'XEvaluate' 'rownames(myMatrix)') (⊃⎕wi 'XEvaluate'
'colnames(myMatrix)') (⎕wi 'XEvaluate' 'dim(myMatrix)')
.→---------------------.
|.→--------..→---..→--.|
|↓FirstRow |↓Col1||2 3||
||SecondRow||Col2|'~--'|
|'---------'|Col3|     |
|           '----'     |
'∈---------------------'
      ⍝ Note that in R, the shape is 2 3
```

Another incompatibility is that R is capable of indexing the matrix by the row and column names as well as by ordinal position in index origin 1 only. APL+Win indexing can be based on index origin 0 or 1 but not by row and column names—unless the row and column names are pre-assigned appropriate ordinal numbers.

```
> myMatrix["SecondRow","Col2"]; # Note the use of comma (not semi-colon) to separate planes.
[1] 4
> myMatrix[2,2]
[1] 4
```

The corresponding APL+Win expressions are:

```
      myMatrix←⍉⎕wi 'XGetSymbol' 'myMatrix' ⍝ Transpose for compatibility
      myMatrix[2;2] ⍝ Index origin 1
4
      SecondRow←2 ⋄ Col2←2
      myMatrix[SecondRow;Col2]
4
```

Bear in mind the following: R holds values and attributes for its objects. APL+Win does not have attributes for its value objects. In order for APL+Win to hold values and attributes in the same object name, it will need to use a nested array. However, there is no unique way (natural positional order) to organise the values and attributes.

Therefore, an expedient solution is for APL+Win to work with values alone and to rely on R to manage the co-existence of values and attributes.

```
      ⊃⎕wi 'XEvaluate' 'capture.output(myMatrix)'
          Col1 Col2 Col3
FirstRow     1    3    5
SecondRow    2    4    6
```

A very welcome bonus with this result is that R served an APL+Win nested vector; this introduces a handy enhancement in the data acquisition and transfer process.

---

[11] For now, ignore the transposition; this is discussed below.

Some attributes, such as row and column names must be unique when applicable to data frames but not when applicable to matrices. Recurring names can cause unexpected behaviour; for example, when using a row or column name for indexing a matrix, R only returns the first matching row or column by default. Consider this example.

```
> myMatrix<-matrix(seq(1:12),ncol=4);
> rownames(myMatrix)<-c('One','Inb','One');
> colnames(myMatrix)<-c('Col1','Col2','Col3','Col4');
> myMatrix;
    Col1 Col2 Col3 Col4
One    1    4    7   10
Inb    2    5    8   11
One    3    6    9   12
> myMatrix['One',]; # Returns first row with name 'One'
Col1 Col2 Col3 Col4
   1    4    7   10
> myMatrix[rownames(myMatrix)=='One',]; # This returns both rows
    Col1 Col2 Col3 Col4
One    1    4    7   10
One    3    6    9   12
```

From APL+Win:

```
      ⎕wi 'XEvaluate' "t(myMatrix[rownames(myMatrix)=='One',]);"
 1 4 7 10
 3 6 9 12
```

Note the following:

- XEvaluate evaluates the first statement only and ignores subsequent ones silently.
- Any variable created within R—from the console or by running a script—needs to be transposed when read into APL+Win.
- Any variable created by APL+Win—using XSetSymbol—within R does not need transposition when read back into APL: the value is transposed inwards and then again outwards, thereby cancelling the effect.

2.2.8. R homogeneous data structures with APL+Win

In order to eliminate character set encoding issues when dealing with naming conventions between R and APL+Win, it is advisable to restrict names to the typewriter keys only, that is A...Z, a...z, and 0...9. Avoid confusion: stick to the APL+Win naming convention and avoid the R flexibility where any name is acceptable, and, do not use any R reserved word as an identifier.

The specification of literal attributes from APL+Win may involve subtle difficulties since APL+Win pads every element of a string array with trailing spaces. Use nested vectors rather than arrays of rank 2 in order to avoid unexpected behaviour.

An R data structure comprises of basic data with attributes beyond rank, length, and shape. Some attributes are read-only, for example, *length*. Other attributes have default values, which can be overridden.; such attributes do not impinge on numerical manipulation of the values. However, they usually have a bearing on the graphical representation of those values.

R data structures lies at the heart of visualisation in R.

2.2.8.1. Vector

A vector is the basic data type of the R language; it is an ordered collection of numeric or character or logical values. All other data structures are derived from a vector by the specification of appropriate attributes.

```
    ∇ SendVector;aplVar;aplVarR
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ ⎕wself is ⍙rfs created by function RFServer
[3]   aplVar←0.01×100⌊10?19021954
[4]   aplVarR←(⍷⍕1⌽')c(',¯1↓⍷aplVar,¨',')~' '
[5]   ⎕wi 'XSetSymbol' 'myVec' aplVar ⍝ Allows APL variables to be passed directly
[6]   Rexp←'myVec<-#'
[7]   ⎕wi 'XEvaluateNoReturn' (⎕←'myVecAlt<-#;' InLineExpression aplVarR)
[8]   ⎕wi 'XEvaluateNoReturn' 'myVecDirect<-rnorm(10);'
    ∇

      SendVector
myVecAlt<-c(0.15,0.58,0.85,0.18,0.32,0.7,0.77,0.09,0.33,0.79);
```

The latter line was output by line [7] for visual confirmation of the argument passed in on that line. Note that there are two ways to pass variables into R, seen on lines [5] and [7]. Clearly, the more efficient method is the one on line [5] since it allows APL+Win workspace variables to be passed directly. However, the syntax in line [7] can be useful when creating values in R without requiring them in the APL session and/or using R functions; line [8] demonstrates this approach.

The function InLineExpression is defined as follows:

```
    ∇ Z←L InLineExpression R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ Build inline expression given function (L) and argument (R) e.g.
[3]   R←(⍕R)~' '
[4]   :if 1∈Z←'$'⍷L
[5]       L←⊂L,R
[6]   :elseif 1∈Z←'#'⍷L
[7]       (Z/L)←⊂R
[8]   :elseif 1∈Z←'()'⍷L
[9]      (Z/L)←⊂'(',R
[10]  :elseif 1∈Z←'(,'⍷L
[11]     (Z/L)←⊂'(',R
[12]  :elseif 1∈Z←'""'⍷L
[13]     (Z/L)←⊂'"',R
[14]  :elseif 1∈Z←"''"⍷L
[15]     (Z/L)←⊂"'",R
[16]  :elseif 1∈Z←'[]'⍷L
[17]     (Z/L)←⊂"[",R
[18]  :else
[19]     Z←⊂L
[20]  :endif
[21]  Z←∊L
    ∇
```

This is a generic function that combines R keywords and their argument to produce valid R executable expressions that can be passed in by APL+Win.

The reverse process, that is, the process of recovering R variables into the R session is equally simple.

```
    ∇ ReceiveVector
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ ⎕wself is ⍙rfs created by function RFServer
[3]   ⎕←aplVarfromR←⎕wi 'XGetSymbol' 'myVec'
[4]   ⎕←aplVarfromRAlt←⎕wi 'XEvaluate' 'myVecAlt'
[5]   ⎕←aplVarfromR≡aplVarfromRAlt
[6]   ⎕←⎕wi 'XGetSymbol' 'myVecDirect'
    ∇

      ReceiveVector
0.51 0.95 0.86 0.12 0.98 0.15 0.78 0.55 0.27 0.58
0.51 0.95 0.86 0.12 0.98 0.15 0.78 0.55 0.27 0.58
1
¯0.143310604 ¯0.5417076592 ¯2.517631838 0.609062457 1.065557654 ¯0.7167182047
¯0.4669290882 ¯0.5883314773 0.3246085037 0.19406793
```

The output of lines [3] to 6], that is the four lines of output are shown in sequence.

APL+Win can also receive the results of evaluated expressions from R *without* the need for intermediate assignment.

```
    ∇ SendReceiveVectorExpression
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ ⎕wself is ⍙rfs created by function RFServer
[3]   ⎕←⎕wi 'XEvaluate' 'seq(from = 1, to=5, by=0.25)'
[4]   ⎕←⎕wi 'XEvaluate' 'sample(sample(5),10,replace=TRUE)'
    ∇

      SendReceiveVectorExpression
1 1.25 1.5 1.75 2 2.25 2.5 2.75 3 3.25 3.5 3.75 4 4.25 4.5 4.75 5
5 3 4 5 1 4 4 2 5 1
```

Line [3] enumerates the range 1 to 5 in steps of 0.25. Line [4] returns a vector of ten elements with replication from a vector of five elements.

Of course, any vector received from R retains its type.

```
    ∇ Z←ReceiveVectorType
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ ⎕wself is ⍙rfs created by function RFServer
[3]   Z←⊂(100×⎕wi 'XEvaluate' 'rnorm(5);')
[4]   Z←Z,⊂⎕wi 'XEvaluate' 'LETTERS[1:26];'
```

```
[5]   Z←Z,⊂⎕wi 'XEvaluate' 'sample(c(TRUE,FALSE),6,replace=TRUE);'
    ∇

      NValue←ReceiveVectorType
      1×1⊃NValue ⍝ Numeric
99.56755922 ¯13.16180569 ¯10.52660973 ¯139.5063174 26.20100761
      2⊃NValue ⍝ Character
 A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
      3⊃NValue ⍝ Logical
1 1 1 1 1 0
```

Please note:

- Character vectors in R come across as nested values in APL+Win.
- Logical values from R come across as 0 or 1 in APL+Win.
- R has no concept of scalar values; scalars are vectors of length 1.

2.2.8.1.1. Familiar vector operations

R has sorting features comparable with APL+Win.

```
> y<-c(sample(1:100,8)) # 8 random numbers
between 1 to 10
> y
[1] 97  3 94 34 15  9 49 69
> sort(y) # Ascending order
[1]  3  9 15 34 49 69 94 97
> order(y) # Grade-up
[1] 2 6 5 4 7 8 3 1
> rev(order(y)) # Grade-down
[1] 1 3 8 7 4 5 6 2
> y[rev(order(y))] # Descending order
[1] 97 94 69 49 34 15  9  3
> min(y);
[1] 3
> max(y);
[1] 97
> ceiling(y);
[1] 97  3 94 34 15  9 49 69
> floor(y);
[1] 97  3 94 34 15  9 49 69
> cumsum(y)
[1]  97 100 194 228 243 252 301 370
> cumprod(y)
[1]  97 291 27354 930036 13950540
[6]  125554860   6152188140 424500981660
> cummin(y)
[1] 97  3  3  3  3  3  3  3
> cummax(y)
[1] 97 97 97 97 97 97 97 97
```

```
      y←⎕wi 'XGetSymbol' 'y'
      y
97 3 94 34 15 9 49 69
      y[⍋y]
3 9 15 34 49 69 94 97
      ⍋y
2 6 5 4 7 8 3 1
      ⍒y
1 3 8 7 4 5 6 2
      y[⍒y]
97 94 69 49 34 15 9 3
      ⌊/y
3
      ⌈/y
97
      ⌈y
97 3 94 34 15 9 49 69
      ⌊y
97 3 94 34 15 9 49 69
+\y
97 100 194 228 243 252 301 370
×\y
97 291 27354 930036 13950540 125554860
6152188140 4.245009817E11
      ⌊\y
97 3 3 3 3 3 3 3
      ⌈\y
97 97 97 97 97 97 97 97
```

```
> scale(y)
           [,1]
[1,]  1.35874458
[2,] -1.15794489
[3,]  1.27842470
[4,] -0.32797283
[5,] -0.83666538
[6,] -0.99730513
[7,]  0.07362655
[8,]  0.60909240
attr(,"scaled:center")
[1] 46.25
attr(,"scaled:scale")
[1] 37.35065
> fivenum(y)
[1]  3.0 12.0 41.5 81.5 97.0
> table(y); #frequency counts of entries
y
 3  9 15 34 49 69 94 97
 1  1  1  1  1  1  1  1
```

R has several other functions with no direct equivalent—you need to consult the online help file (? function) to investigate what they do.

#Tukey five numbers min, lower hinge, median, upper hinge, max

Finally, all the standard statistical functions are available out of the box; these are accessible from

```
> mean(y);
[1] 46.25
> median(y);
[1] 41.5
```

APL+Win.

```
      ⎕wi 'XEvaluate' 'fivenum(y)'
3 12 41.5 81.5 97
      ⎕wi 'XEvaluate' 'mad(y)'
44.478
```

```
> sum(y);
[1] 370
> var(y); #produces the variance covariance
matrix
[1] 1395.071
> sd(y);  #standard deviation
[1] 37.35065
> mad(y); #(median absolute deviation)
[1] 44.478
```

2.2.8.1.2. Unfamiliar vector operations

Some aspects of vector operations can become highly confusing for an APL developer.

```
> vecA<-c(2,4,3);
> vecB<-c(4,6,7,8,9,10);
> vecC<-vecA+vecB; # Length(s) are conformable in R but not in the APL sense
> length(vecC);
[1] 6
> vecC;            # Re-uses vecA
[1]  6 10 10 10 13 13
> vecC[20]<--1;    # Can assign to an index that does not exist!
> vecC;
 [1]  6 10 10 10 13 13 NA NA NA NA NA NA NA NA NA NA NA NA -1
> # NA is for missing value; vecC has been padded!
```

2.2.8.2. Matrix

An R matrix is a vector with a dimension attribute and is always of rank 2, that is, is always 2-dimensional.

```
    ∇ SendMatrix
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ ⎕wself is ⍋rfs created by function RFServer
[3]   ⎕wi 'XSetSymbol' 'aplMat' (2 3⍴1 2 3 4 5 6)
[4]   ⎕wi 'XEvaluateNoReturn'  'aplMatSyn1<-matrix(c(1,2,3,4,5,6),ncol=2,byrow=FALSE);'
[5]   ⎕wi 'XEvaluateNoReturn'  'aplMatSyn2<-matrix(c(1,2,3,4,5,6),nrow=2,byrow=TRUE);'
[6]   ⎕wi 'XEvaluateNoReturn' 'aplMatSyn3<-c(1,2,3,4,5,6);'  ⍝ Step 1
[7]   ⎕wi 'XEvaluateNoReturn' 'dim(aplMatSyn3)<-c(2,3);' ⍝ Step 2
    ∇
```

This function illustrates three methods for passing a matrix into R.

Line [3] illustrates the simplest method for the following reasons:

- The value is passed from an APL variable or expression.
- Sending the value to R transposes it and receiving it back also transposes it: therefore, the effect cancels out.
- Optionally, the APL value can be transposed, where necessary, to achieve the correct scenario within R.

```
      (2 3⍴1 2 3 4 5 6)≡⎕wi 'xGetSymbol' 'aplMat'
1
```

Line [4] demonstrates the second method.

- The values are passed in as inline code; this makes for a rather cumbersome approach to building the expression, especially where the number of elements is large.
- The collating sequence is coerced into the same order as APL+Win, namely, column-wise.
- The problems of transposition is circumvented by an implicit transposition, namely, the number of rows is specified as the number of columns.

```
      (2 3⍴1 2 3 4 5 6)≡⎕wi 'xGetSymbol' 'aplMatSyn1'
1
```

Line [5] demonstrates the third method; this extends to line [6] since the XEvaluateNoReturn method acts upon a single statement at a time and this method involves two statements.

- Line [5] creates a vector of values.
- Line [6] applies the attribute dimension to the values.

Visually and syntactically, the values are identical within APL+Win.

```
      (2 3⍴1 2 3 4 5 6) (⍉⎕wi 'xGetSymbol' 'aplMatSyn2')
  1 2 3    1 2 3
```

```
  4 5 6    4 5 6
     (2 3⍴1 2 3 4 5 6)≡(⍎⎕wi 'xGetSymbol' 'aplMatSyn2')
1
```

However, R reports them as different—note the type of the values reported below.

```
> any(aplMat !=aplMatSyn2) # one is 3 2 the other 2 3
Error in aplMat != aplMatSyn2 : non-conformable arrays
> dim(aplMat) # Verify dimension
[1] 3 2
> dim(aplMatSyn2)# Verify dimension
[1] 2 3
> identical(aplMat,t(aplMatSyn2)) # Transpose. Check if they match
[1] FALSE
> any(aplMat != t(aplMatSyn2)) # Is any element unequal
[1] FALSE
> c(typeof(aplMat),class(aplMat)) # Where is the difference?
[1] "integer" "matrix"
> c(typeof(t(aplMatSyn2)),class(aplMatSyn2))
[1] "double" "matrix"
```

In conclusion, the optimal method is to pass variables by value from R and APL+Win.

- Passing a value from an APL+Win expression or workspace variable makes for a simpler syntax and is more reliable. However, remember that the value gets transposed into and out of either environment.
- Passing a value as an argument to a function is much easier by name than by value; this is only possible if the variable exists in that environment; refer to the call to the identical function above.

The considerations for Boolean values are identical to that of passing numeric values. However, R will see them as TRUE or FALSE; I believe these are predefined constants that hold 1 or 0 as in APL+Win.

Character data passing is also straightforward.

```
      ⎕wi 'XSetSymbol' 'nl23' ('R' ⎕nl 2 3)
```

However, there are some subtleties:

```
      ⎕wi 'XEvaluate' 'c(length(nl23),typeof(nl23),dim(nl23),class(nl23))'
 6 character 6 array
```

Remember that everything is a vector by default; the APL+Win enumeration returns a character array of rank 2. Unlike APL which counts every column in the second dimension, R will see a character matrix as comprising of a single column. Therefore the correct syntax is:

```
      ⎕wi 'XEvaluateNoReturn' 'nl23Alt<-as.matrix(nl23,ncol=1);'
```

2.2.8.2.1. Other matrices features

There are other useful techniques—lacking in APL+Win—for constructing matrices.

```
> x<-c(1,2,4,8,16); # A numeric vector of 4 elements
> y<-c(1:10);       # A numeric vector of 10 elements
> mat=cbind(x,y);   # Concatenate on the y axis
> mat2=rbind(x,y);  # Concatenate on the x axis
```

The expressions executed successfully. Let's examine the results In APL+Win.

```
      ⍎⎕wi 'XGetSymbol' 'mat'
 1  1
 2  2
 4  3
 8  4
16  5
 1  6
 2  7
 4  8
 8  9
16 10
```

Since the matrices were created within R, they must be transposed—within R or APL+Win--when read back into APL.

The expression that created this matrix will execute successfully if and only if both vectors are the same length or the length of one vector is an exact multiple of the length of the other vector.

```
      ⎕wi 'XEvaluate' 'dim(mat)'
10 2
```

```
      ⎕wi 'XEvaluate' 't(mat2)'
 1  2  4  8 16  1  2  4  8 16
 1  2  3  4  5  6  7  8  9 10
```

```
      ⎕wi 'XEvaluate' 'dim(mat2)'
2 10
```

R also has a number of built-in functions for basic computations on matrices.

```
      ⎕wi 'XEvaluateNoReturn' 'diaMat<-diag(5)'
      (⍎⎕wi 'XGetSymbol' 'diaMat') (5 5⍴6↑1)
1 0 0 0 0    1 0 0 0 0
0 1 0 0 0    0 1 0 0 0
0 0 1 0 0    0 0 1 0 0
0 0 0 1 0    0 0 0 1 0
0 0 0 0 1    0 0 0 0 1
```

```
      myMat←4 5⍴20?100
      ⎕wi 'XSetSymbol' 'myMat' (⍎myMat
      (+/myMat)÷0⌷⍴myMat
41.6 44.6 60.8 60.2
      ⎕wi 'XEvaluate' 'rowMeans(myMat)'
41.6 44.6 60.8 60.2
      (+⌿myMat)÷↑⍴myMat
68 50.75 55.5 37.25 47.5
      ⎕wi 'XEvaluate' 'colMeans(myMat)'
68 50.75 55.5 37.25 47.5
⍝ Try also: nrow(), ncol(), sum(), mean(),
dim(), det(), solve()
```

R can assign names to both the rows and columns of matrices; the names can then be used to index the cells.

```
> thisMat<-c(sample(1:1000,30)); # A vector of 30 elements
> dim(thisMat)<-c(5,6);          #  Set the dimensions to 5 rows 6 columns
> rownames(thisMat)<-c("Y1","Y2","Y3","Y4","Y5"); # Give each row a name
> colnames(thisMat)<-c("1980","1981","1982","1983","1984","1985"); # Give each column a name
> thisMat; # Verify how it looks
   1980 1981 1982 1983 1984 1985
Y1  946  947  404  549   57  873
Y2  987  617  992  120  361  478
Y3  297  787  389  650  859  897
Y4  445  588  181  387  735  753
Y5  108  956  855  694  172  217
> thisMat[,"1984"]; # Results retains attributes
 Y1  Y2  Y3  Y4  Y5
 57 361 859 735 172
> thisMat[thisMat %in% c(588,855)] # Conditional Selection 1
[1] 588 855>
> thisMat[thisMat>900] # Conditional Selection 2
[1] 946 987 947 956 992
> thisMat[thisMat>900]<-(-1) # Conditional Assignment
> thisMat;
   1980 1981 1982 1983 1984 1985
Y1   -1   -1  404  549   57  873
Y2   -1  617   -1  120  361  478
Y3  297  787  389  650  859  897
Y4  445  588  181  387  735  753
Y5  108   -1  855  694  172  217
```

Now APL+Win can retrieve the matrix:

```
      ⎕←thisMat←⍎⎕wi 'XGetSymbol' 'thisMat' ⍝ Row/Column names do not come across
 ¯1  ¯1 404 549  57 873
 ¯1 617  ¯1 120 361 478
297 787 389 650 859 897
445 588 181 387 735 753
108  ¯1 855 694 172 217
```

Note that the negative numbers now have high minus. Although the row and column names can be retrieved separately, they cannot be used for indexing within APL+Win.

```
      (⎕wi 'XEvaluate' 'rownames(thisMat)') (⎕wi 'XEvaluate' 'colnames(thisMat)')
Y1 Y2 Y3 Y4 Y5   1980 1981 1982 1983 1984 1985
```

However, they can be inside R.

```
      ⎕←thisMat←⍎⎕wi 'XGetSymbol' 'thisMat' ⍝ Row/Column names do not come across
 ¯1  ¯1 404 549  57 873
 ¯1 617  ¯1 120 361 478
297 787 389 650 859 897
445 588 181 387 735 753
108  ¯1 855 694 172 217
     (⎕wi 'XEvaluate' 'thisMat["Y2","1983"]') (thisMat[2;4])
     (⎕wi 'XEvaluate' 'thisMat["Y2","1983"]') (thisMat[2;4])
120 120
     (⎕wi 'XEvaluate' 'thisMat["Y2",c("1981","1984")]') (thisMat[2;2 5])
617 361  617 361
```

Compare the result of the latter statement executed in R:

```
> thisMat["Y2",c("1981","1984")]; # The result displays with the column headers
1981 1984
 617  361
```

Finally, the APL+Win inner product is also available in R:

```
      vec1←5?100 ⋄ vec2← 3?10
      (⎕wi 'XSetSymbol' 'vec1' vec1) (⎕wi 'XSetSymbol' 'vec2' vec2)
      (vec1∘.+vec2) (⍎⎕wi 'XEvaluate' 'outer(vec1,vec2,"+")')
 65 57 62    65 57 62
 14  6 11    14  6 11
 86 78 83    86 78 83
 88 80 85    88 80 85
 41 33 38    41 33 38
```

2.2.8.3. Array

An R array is an ordered collection of values which has more than 2 dimensions. I found that the simplest option is to pass values into either environment by value; therefore there is no need to investigate other options.

```
      aplRank3←2 3 4⍴24?1999
      ⎕wi 'XSetSymbol' 'aplRank3R' aplRank3
      aplRank3≡⎕wi 'XGetSymbol' 'aplRank3R'
1
      ⎕wi 'XEvaluate' 'range(aplRank3R)'
34 1729
      (⌊/,aplRank3),⌈/,aplRank3
34 1729
```

Equally, the observations regarding character and Boolean matrices apply to arrays of the same type.

2.2.8.4. Factor and Table

APL+Win does not have a direct equivalent to these data structures. In fact, this data structure holds information—that is, data derived from data—rather than raw data. This structure is easily simulated within APL. The function should be self-explanatory as it uses standard APL constructions.

If necessary, run the following function in trace mode and examine each line of code.

```
    ∇ Z←FactorInAPL;depts;freq;dept;factor;q
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   dept←(3↑¨'Finance' 'Marketing' 'IT' 'Administration' 'Investment' 'HR'
'Actuarial')~¨' '
[3]   dept←((dept⍳dept)=⍳⍴dept)/dept ⍝ Ensure that each nominal value is unique
[4]   dept←dept[⍋⎕av⍳⊃dept]
[5]   freq←(⍴dept)?⍳30               ⍝ 30 staff allocated arbitrarily to depts
[6]   depts←dept[freq+⎕io]
[7]   depts←depts[⍋⎕av⍳⊃depts]
[8]   factor←dept⍳depts ⍝ Ordinal value of dept
[9]   factor←+⌿factor∘.=((factor⍳factor)=⍳⍴factor)/factor
[10]  Z←('From APL+Win') (⊃(dept[order]) (factor[order]))
[11]  q←'myFactor<-factor(c(#));' InLineExpression CSVWrap depts
[12]  ⎕wi 'XEvaluateNoReturn' q
[13]  ⎕wi 'XEvaluateNoReturn' 'myTable<-table(myFactor);'
[14]  Z←⊃Z (('From R') (⊃1↓⎕wi 'XEvaluate' 'capture.output(myTable);'))
    ∇
```

The utility function is:

```
    ∇ R←CSVWrap R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   R←¯1↓∊'"',¨R,¨⊂'",'
    ∇

      FactorInAPL
 From APL+Win  Mar HR Act Inv Fin IT Adm
                 4  4   4   4   5  4   5

 From R      Act Adm Fin  HR Inv  IT Mar
               4   5   5   4   4   4   4
```

Note that the order of the results from APL+Win and R are different. Since R has built-in functionality to handle factors and tables, it is recommended that these data structures be acquired from R. APL+Win simply needs to supply the raw data. Optionally, R can also order the nominal and ordinal values.

2.2.9. Homogeneous data generation

The basic data types are literals, numbers, and dates. In APL+Win, an identifier that holds data of the same type are generally simple, that is, of depth 1. Like APL, R is a type-inferred language; that is, it is not possible to declare the type of identifiers. Identifiers come to exist upon assignment. Homogeneous identifiers hold data of the same type.

Except for the *list* data structure, the R data structures comprise of data of several types; however, each component is of the same type. As R is a statistical—that is, sampling is a recurring theme—language, it is appropriate to illustrate homogeneous data generation techniques in R.

- R has several structures for holding homogeneous data—that is, data of the same type. Given any structure, an identifier's structure and data type can be established..
- R's default data structure is a vector. Note that it does not have a scalar; a scalar is simply a vector of length 1.
- Each data structure has its own set of attributes; some of vector attributes are demonstrated here.

Any vector may be transformed into another R data structure, by modifying its attributes.

2.2.9.1. Numbers

The following creates a numeric vector of five numbers between 0 and 1. The parameters of the underlying function, *runif*, may be used to create numbers with other precision.

```
> myNumericVector<-runif(5); # Five random numbers between 0 and 1
> myNumericVector; # See its contents
[1] 0.3603916 0.3534813 0.7325422 0.3799669 0.3949394
> class(myNumericVector); # Structure
[1] "numeric"
> typeof(myNumericVector);# Type
[1] "double"
> mode(myNumericVector);  # Storage
[1] "numeric"
```

The numeric vector can be transformed into a matrix.

```
> myArray<-as.matrix(myNumericVector)
> dim(myArray)
[1] 5 1
```

2.2.9.2. Literals

The following illustrates an anonymous function, that is, lambda expression that generates a vector of five 'random' strings, each between 5 to 8 characters in length.

```
> myCharacterData<-(function(i){ret="";
+              for (j in 1:i){
+                  thisRet=paste(sample(c(LETTERS,
letters),sample(5:8,1),replace=TRUE),collapse="");
+                  ret<-c(ret,thisRet);
+             }
+             return(ret[2:length(ret)])
+             })(5); # Anonymous function to generate 5 random strings 5-8 chars in length
> myCharacterData; # See its contents
[1] "nkJhYpko" "GDOjUa"   "sqkfTjbW" "Sjfgi"    "eEaEPrvJ"
> class(myCharacterData); # Structure
[1] "character"
> typeof(myCharacterData);# Type
[1] "character"
> mode(myCharacterData);  # Storage
[1] "character"
```

This character can be transformed into another data structure.

```
> myList<-as.list(myCharacterData);
> length(myList);
[1] 5
```

2.2.9.3. Dates

The following creates a vector of 'random' five dates.

```
> myDateVector<-as.Date(runif(5,min=12000,max=23400),origin="1900-01-01");# Five random dates
> myDateVector; #See its content
[1] "1942-07-11" "1937-12-28" "1943-11-19" "1963-07-16" "1948-11-27"
> class(myDateVector); # Structure
[1] "Date"
> typeof(myDateVector);# Type
[1] "double"
> mode(myDateVector);  # Storage
```

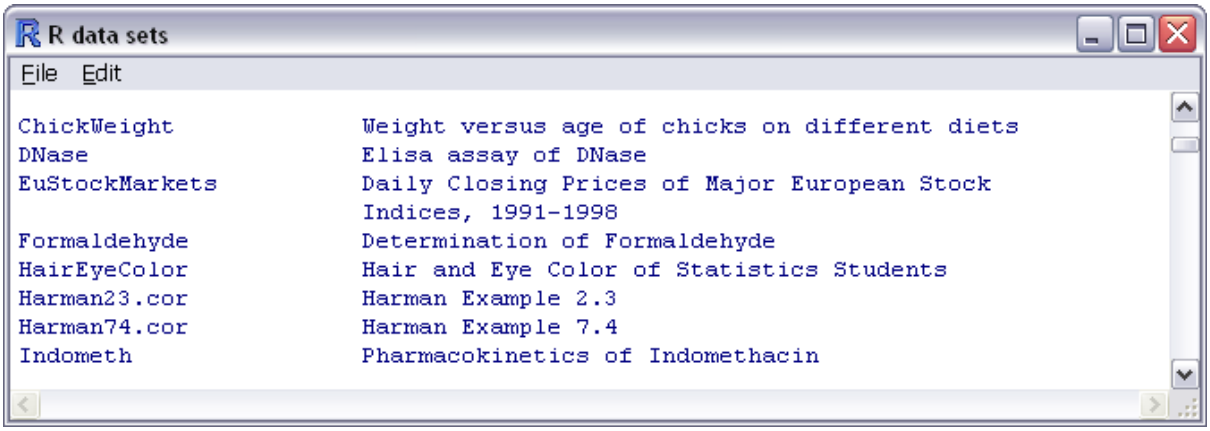
```
[1] "numeric"
```

The date vector can be coerced into another data structure.

```
> as.factor(myDateVector)
[1] 1942-07-11 1937-12-28 1943-11-19 1963-07-16 1948-11-27
Levels: 1937-12-28 1942-07-11 1943-11-19 1948-11-27 1963-07-16
```

2.2.10. Homogeneous data coercion

The basic installation of R provides a number of data structures that are handy for exploration of R features. The function *data()* exposes the full list—for browsing—in a graphical interface.



The following illustrates the retrieval process for any of the data structures into the session:

```
> data(AirPassengers)
> AirPassengers
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
1949 112 118 132 129 121 135 148 148 136 119 104 118
1950 115 126 141 135 125 149 170 170 158 133 114 140
1951 145 150 178 163 172 178 199 199 184 162 146 166
1952 171 180 193 181 183 218 230 242 209 191 172 194
1953 196 196 236 235 229 243 264 272 237 211 180 201
1954 204 188 235 227 234 264 302 293 259 229 203 229
1955 242 233 267 269 270 315 364 347 312 274 237 278
1956 284 277 317 313 318 374 413 405 355 306 271 306
1957 315 301 356 348 355 422 465 467 404 347 305 336
1958 340 318 362 348 363 435 491 505 404 359 310 337
1959 360 342 406 396 420 472 548 559 463 407 362 405
1960 417 391 419 461 472 535 622 606 508 461 390 432
> typeof(AirPassengers);
[1] "double"
> class(AirPassengers);
[1] "ts"
```

The sample data sets provide a readily available source for acquisition into APL+Win. Alternatively, generate random homogeneous data components and assemble them into data structures. With the data generated above, a data frame is created as follows:

```
> myDF<-data.frame(myNumericVector,myDateVector,row.names=myCharacterData)
> myDF;
         myNumericVector myDateVector
nkJhYpko       0.3603916   1942-07-11
GDOjUa         0.3534813   1937-12-28
sqkfTjbW       0.7325422   1943-11-19
Sjfgi          0.3799669   1963-07-16
eEaEPrvJ       0.3949394   1948-11-27
```

Incidentally, in order to retrieve data of type date from R, the data must be converted to character.

```
      ⊃⎕wi 'XEvaluate' 'as.character(myDateVector)'
1942-07-11
1937-12-28
1943-11-19
1963-07-16
1948-11-27
```

2.2.11. R heterogeneous data structures with APL+Win

The data structures that lie at the heart of the power features of R are of mixed types, and each type has its own set of attributes. These structures can be constructed from APL+Win in steps. However, the end result is not always directly readable in APL+Win. Why would you want to?

During the investigation of the data structures that are available with R, a sensible step is to examine the attributes of the data structure. For example, for the data frame constructed in the next section, the attributes are:

```
> attributes(classResults)
$names
[1] "subjEnglish"  "gradeEnglish" "subjFrench"   "gradeFrench"

$row.names
 [1] "Student1"  "Student2"  "Student3"  "Student4"  "Student5"  "Student6"
 [7] "Student7"  "Student8"  "Student9"  "Student10"

$class
[1] "data.frame"
```

Note the literals beginning with the $ sign. These can be used to query individual attributes.

```
> names(classResults)
[1] "subjEnglish"  "gradeEnglish" "subjFrench"   "gradeFrench"
```

All data structures do not have attributes.

APL+Win can construct any data structure one step at a time.

```
        ∇ BuildHeteregeneousDataStructure
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ Step1: Create each component of the list
[3]   ⎕wi 'XSetSymbol' 'comp1' (1 2 3 4)
[4]   ⎕wi 'XSetSymbol' 'comp2' (⍉2 3⍴17 2 3 4 18 19)
[5]   ⎕wi 'XSetSymbol' 'comp3' ("Ajay" "Askoolum")
[6]   ⍝ Step2: Create the data structure
[7]   ⎕wi 'XEvaluateNoReturn' 'myListx<-list(comp1=comp1,comp2=comp2,comp3=comp3);'
[8]   ⍝ Step3: Remove the components of the data structure ... the data structure is
unaffected by this
[9]   ⎕wi 'XEvaluateNoReturn' 'rm(list=c("comp1","comp2","comp3"));'
    ∇
      ⎕wi 'XEvaluate' 'exists("myListx");' ⍝ Does not exist
0
      BuildHeteregeneousDataStructure      ⍝ Build it
      ⎕wi 'XEvaluate' 'exists("myListx");' ⍝ Check that it exists
1
      ⎕wi 'XEvaluate' 'sapply(c("comp1","comp2","comp3"),exists);' ⍝ Remove the components
0 0 0
      ⎕wi 'XGetSymbol' 'myListx' ⍝ Fails to bring the list across
 1 17 Ajay
```

Given that the data structure does not come across, there are several strategies for visually examining its contents from within APL+Win: note, this shows the content of each component and not its attributes.

```
      ⊃⎕wi 'XEvaluate' 'sapply(names(myListx),function(x)
paste(x,paste(myListx[[x]],collapse=" ")));'
comp1 1 2 3 4
comp2 17 4 2 18 3 19
comp3 Ajay Askoolum
```

Or, try these expressions—note they show the content and attribute of each component—from within R[12]:

```
> str(myListx);
List of 3
 $ comp1: int [1:4(1d)] 1 2 3 4
 $ comp2: int [1:2, 1:3] 17 4 2 18 3 19
 $ comp3: chr [1:2(1d)] "Ajay" "Askoolum"
> dput(myListx);
structure(list(comp1 = structure(1:4, .Dim = 4L), comp2 = structure(c(17L,
```

[12] Do not evaluate either of these expressions from APL+Win: they dump a binary pattern of their result respective result.

```
4L, 2L, 18L, 3L, 19L), .Dim = 2:3), comp3 = structure(c("Ajay",
"Askoolum"), .Dim = 2L)), .Names = c("comp1", "comp2", "comp3"
))"
```

The optimal approach is to write the executable definition of the data structure to a text file, as a script that R can execute.

2.2.11.1. Data Frames

A data frame holds tabular data, has rank 2, all columns have the same length but can be of different types; it is the primary date structure in R and, visually, it is an excel sheet or a relational table..

The following function creates a data frame in R.

```
    ∇ DataFrame;studentName;subjEnglish;subjFrench;gradeEnglish;gradeFrench
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ Incremental build-up of R data structure from APL+Win
[3]   studentName←(⊂'Student'),¨⍕¨+\10/1
[4]   subjEnglish←10?100
[5]   subjFrench←10?100
[6]   gradeEnglish←(,¨'X?UFEDCBA★')[⎕io++/subjEnglish∘.>19 29 39 49 59 69 79 89 92]
[7]   gradeFrench←(,¨ 'X?UFEDCBA★')[⎕io++/subjFrench∘.>19 29 39 49 59 69 79 89 92]
[8]   ⎕wi 'XSetSymbol' 'subjEnglish' subjEnglish
[9]   ⎕wi 'XSetSymbol' 'subjFrench' subjFrench
[10]  ⎕wi 'XSetSymbol' 'gradeEnglish' gradeEnglish
[11]  ⎕wi 'XSetSymbol' 'gradeFrench' gradeFrench
[12]  ⎕wi 'XSetSymbol' 'studentName' studentName
[13]  ⎕wi 'XEvaluateNoReturn' 'classResults<-
data.frame(subjEnglish,gradeEnglish,subjFrench,gradeFrench);'
[14]  ⎕wi 'XEvaluateNoReturn' 'rownames(classResults)<-studentName;'
[15]  ⍝ Tidy up by removing intermediate variables
[16]  ⎕wi 'XEvaluateNoReturn' 'rm(studentName)'
[17]  ⎕wi 'XEvaluateNoReturn' 'rm(list=names(classResults));' ⍝ Variables in data.frame
    ∇
```

In R, the data frame *classResults* can be examined.

```
> classResults
          subjEnglish gradeEnglish subjFrench gradeFrench
Student1           88            B         41           F
Student2           19            X         35           U
Student3            6            X         93           *
Student4           93            *          2           X
Student5           91            A         19           X
Student6           17            X         56           E
Student7           81            B         92           A
Student8           99            *         83           B
Student9           72            C          3           X
Student10          55            E         13           X
> class(classResults);
[1] "data.frame"
```

2.2.11.2. Time Series

A bare basic time series data structure comprises of a series of values applicable from a starting time value that specifies the year. The values may be applicable to a frequency; usually this will be one of monthly (12), quarterly (4), or yearly (1), the default. When frequency is 4 and 12, the labels (Qtr1, Qtr2 … Qtr4) and (Jan, Feb … Dec) are applied automatically.

2.2.11.2.1. Time Series - yearly

Any given vector of raw values can be coerced into a time series data structure; there are several ways of doing so:

```
> values <-c(12,34,54,67,87,34,65,34,32,12,43,89,23,54,23,43,13);
> valuesTS1<-ts(values,start=2000);
> valuesTS2<-ts(values,c(2000),,1);
> valuesTS3<-ts(values,c(2000));
> identical(valuesTS1,valuesTS2,valuesTS3);
[1] TRUE
>
```

The expressions I have used do not specify the end or frequency of the time series; these are implicitly worked out by R. This technique sometimes avoids the introduction of unnecessary logic flaws and has no impact of the result. The full specification of any of the time series variables can be examined:

```
> str(valuesTS2)
 Time-Series [1:17] from 2000 to 2016: 12 34 54 67 87 34 65 34 32 12 ...
> dput(valuesTS4)
structure(c(12, 34, 54, 67, 87, 34, 65, 34, 32, 12, 43, 89, 23,
54, 23, 43, 13), .Tsp = c(2000, 2016, 1), class = "ts")
```

2.2.11.2.2. Time Series – yearly by quarter

In order to specify that the values apply from a specific year and quarter, a different syntax is required. The following example specifies that the series starts with the fourth quarter of 2012.

```
> valuesTS1Q<-ts(values,start=c(2000,4),frequency=4);
> valuesTS1Q;
     Qtr1 Qtr2 Qtr3 Qtr4
2000                   12
2001   34   54   67   87
2002   34   65   34   32
2003   12   43   89   23
2004   54   23   43   13
> dput(valuesTS1Q);
structure(c(12, 34, 54, 67, 87, 34, 65, 34, 32, 12, 43, 89, 23,
54, 23, 43, 13), .Tsp = c(2000.75, 2004.75, 4), class = "ts")
```

Values apply from year 2000, 4th quarter; the preceding quarters will have null values. Depending on the length of values, there may be trailing null values.

2.2.11.2.3. Time Series – yearly by month

In order to specify that the values apply from a specific year and month, a variation on the syntax is required.

```
> valuesTS1M1<-ts(values,start=c(2000,7),frequency=12); # July 2000
> valuesTS1M2<-ts(values,c(2000,7),,12);
> identical(valuesTS1M1,valuesTS1M2);
[1] TRUE
> valuesTS1M1;
     Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec
2000                         12  34  54  67  87  34
2001  65  34  32  12  43  89  23  54  23  43  13
> dput(valuesTS1M1);
structure(c(12, 34, 54, 67, 87, 34, 65, 34, 32, 12, 43, 89, 23,
54, 23, 43, 13), .Tsp = c(2000.5, 2001.83333333333, 12), class = "ts")
```

2.2.11.2.4. APL+Win – sending/receiving time series data

Crucially, R time series data is simply a vector of numeric values with other attributes.

```
      ⎕wi¨ (⊂c'XGetSymbol'),¨c¨'valuesTS1' 'valuesTS1Q' 'valuesTS1M1'
 12 34 54 67 87 34 65 34 32 12 43 89 23 54 23 43 13
 12 34 54 67 87 34 65 34 32 12 43 89 23 54 23 43 13
 12 34 54 67 87 34 65 34 32 12 43 89 23 54 23 43 13
```

In R, I used the dput() function to see the data structure; unfortunately, called from APL+Win, this function returns a different result.

```
      ⎕wi 'XEvaluate' 'dput(valuesTS1)' ⍝ Returns raw values in APL+Win, structure in R
12 34 54 67 87 34 65 34 32 12 43 89 23 54 23 43 13
```

A different tact is required in APL+Win:

```
      ⎕wi 'XEvaluate' 'capture.output(valuesTS1)' ⍝ Get dput() in APL+Win
 Time Series: Start = 2000  End = 2016  Frequency = 1   [1] 12 34 54 67 87 34 65 34 32 12
43 89 23 54 23 43 13
```

For this yearly time series variable, the attributes Start, End, and Frequency is now obvious in APL+Win. However, from APL+Win, these appear to be read-only properties.

```
> start(valuesTS1)
[1] 2000    1
> end(valuesTS1)
[1] 2016    1
> frequency(valuesTS1)
[1] 1
```

The read/write property that allows access to all three properties is:

```
> tsp(valuesTS1)
[1] 2000 2016    1
```

This function takes three mandatory numeric arguments, specifying the start, end, and frequency. Except for yearly data, either or both of the first two arguments may contain a fractional part indicating the beginning and end periods respectively. For yearly data, the first two arguments are integer and the third argument is always 1.

It is obvious that a simple vector cannot be passed as an R time series data structure: time series attributes must be specified also. There are two options:

- APL+Win can construct and cause R to evaluate a complete time series expression—which can be a little contrived when the number of values grows—in its own environment.

```
      ⎕wi 'XEvaluateNoReturn' 'myTS<-ts(c(12,23,43,23),start=c(2011,3),frequency=4);'
```

- APL+Win can pass a workspace variable as a vector and then coerce this vector into an R time series structure. Although this method is slightly convoluted, it does have the advantage that the values can be passed from an APL variable and all the attributes must be specified. In the simpler example, only the *start* and not the *end* value was specified; R determined the *end* value implicitly. A simple APL+Win function can calculate all the attributes.

```
    ∇ Z←L GetTSP R;StartYear;StartPeriod;Frequency
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⍝ L = Vector of values, R=StartYear, StartPeriod , Frequency
[3]   ⍝ For yearly time series, StartPeriod and Frequency are always 1
[4]   ⍝ For monthly time series, StartPeriod is [1|2|3|4|5|6|7|8|9|10|11|12] and Frequency
is 12
[5]   ⍝ For quarterly time series, StartPeriod is [1|2|3|4] and Frequency is 4
[6]   (StartYear StartPeriod Frequency)←R
[7]   R←StartYear+(StartPeriod-1)÷Frequency
[8]   R←R,R+(¯1+⍴L)÷Frequency
[9]   R←R,Frequency
[10]  Z←'c(#,#,#);'
[11]  ((Z='#')/Z)←⍕¨R
[12]  Z←⍎⍕¨Z
    ∇
```

The following code creates the same R time series data structure.

```
      aplValues←12 23 43 23 ⍝ No need to render as CSV
      ⎕wi 'XSetSymbol' 'myTS' aplValues ⍝ Vetor is passed in
      ⍝ Coerce myTS into a time series with default attributes
      ⎕wi 'XEvaluateNoReturn' 'myTS<-ts(myTS);'
      ⍝ Set the attributes proper
      ⎕wi 'XEvaluateNoReturn' ("tsp(myTS)<-#;" InLineExpression aplValues GetTSP 2011 3 4)
```

The theme of R pathways continues: the latter expression can be substituted by the following one:

```
⎕wi 'XEvaluateNoReturn' ('attr(myTS,"tsp")<-#;' InLineExpression (aplValues GetTSP 2013 3
4))
```

Note that the attribute *tsp* requires three mandatory arguments, the *start*, *end*, and *frequency* values. The *start* and end *values* can be floating point numbers- the fractional part indicate that the values are partway through the year, specified as the integer part.

Within R, the value is:

```
> myTS
     Qtr1 Qtr2 Qtr3 Qtr4
2011              12   23
2012   43   23
```

Its *tsp* attribute is:

```
> tsp(myTS) # or use attr(myTS,"tsp")
[1] 2013.50 2014.25    4.00
```

In APL+Win, the attributes that is passed is:

```
      aplValues GetTSP 2013 3 4
c(2013.5,2014.25,4);
```

The value of this variable is:

```
      ⎕wi 'XGetSymbol' 'myTS'
12 23 43 23
      ⍝ Note, the attributes do not transfer
      ⍝ Attributes are queried separately
      ⎕wi 'XEvaluate' 'attr(myTS,"tsp")'
2013.5 2014.25 4
```

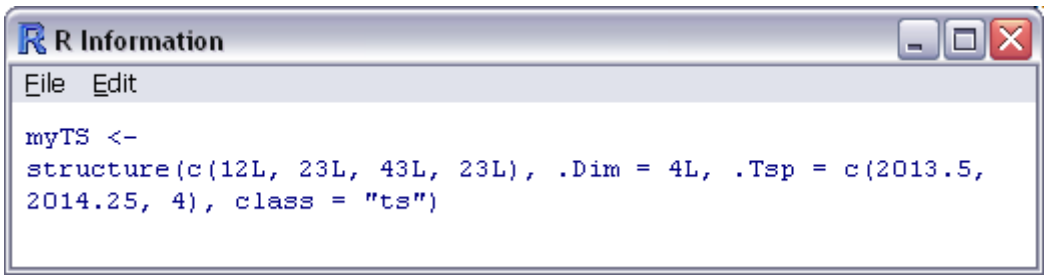
There is a third option for APL+Win: write a script that will create the data structure. R can return an expression (note there is no assignment):

```
> dput(myTS) # Executable expression
structure(c(12L, 23L, 43L, 23L), .Dim = 4L, .Tsp = c(2013.5, 2014.25, 4), class = "ts")
> str(myTS) # Compact display of internal structure
```

```
 'ts' int [1:4(1d)] 12 23 43 23
 - attr(*, "tsp")= num [1:3] 2014 2014 4
```

Or, it can save the full definition of the data structure to a file:

```
> dump(c("myTS"),file="c:/myTS.R"); # file is overwritten automatically
> file.show("c:/myTS.R");
```



Note that R expressions can span multiple lines. R can read this file—that is, re-create the variable—with the following expression:

```
> source("c:/myTS.R");
```

2.2.11.3. List

The list data structure closely corresponds with the APL+Win nested variable type in the sense that it is the only data structure that can hold values of different types.

```
> myList<-list(c(1,2,3,4),matrix(c(17,2,3,4,18,19),ncol=2),c("Ajay","Askoolum"));
> myList;
[[1]]
[1] 1 2 3 4

[[2]]
     [,1] [,2]
[1,]   17    4
[2,]    2   18
[3,]    3   19

[[3]]
[1] "Ajay"     "Askoolum"
```

However, only the first element of each component transfers correctly.

```
      (⎕wi 'XGetSymbol' 'myList') (⎕wi 'XEvaluate' 'myList')
  1 17 Ajay   1 17 Ajay
```

In order to read each component correctly, it must be read iteratively.

```
    ∇ Z←ReadList
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   Z←0/⊂''
[3]   :for i :in +\(⎕wi 'XEvaluate' 'length(myList)')/1 ⍝ R has index origin 1
[4]       Z←Z,⊂⎕wi 'XEvaluate' ('myList[[#]]'  InLineExpression i)
[5]   :endfor
    ∇
     ReadList
 1 2 3 4    17  2  3    Ajay Askoolum
             4 18 19
     ]display ⍴¨ReadList
.→----------.
|.→..→--..→.|
||4||2 3||2||
|'~''~--''~'|
'∈----------'
```

Can you spot the problem? The second component is a matrix and it needs to be transposed; this makes the retrieval process rather convoluted for several reasons.

- The list can contain other lists, and each component can be a different class. In R, every data structure can be reduced to a list.
- Each component may have one or more attributes; as noted, attributes need to be queried independently of the data.

Perhaps, this is the least useful data structure from the point of view of APL+Win. One available method of retrieving the list is to get its internal definition from R:

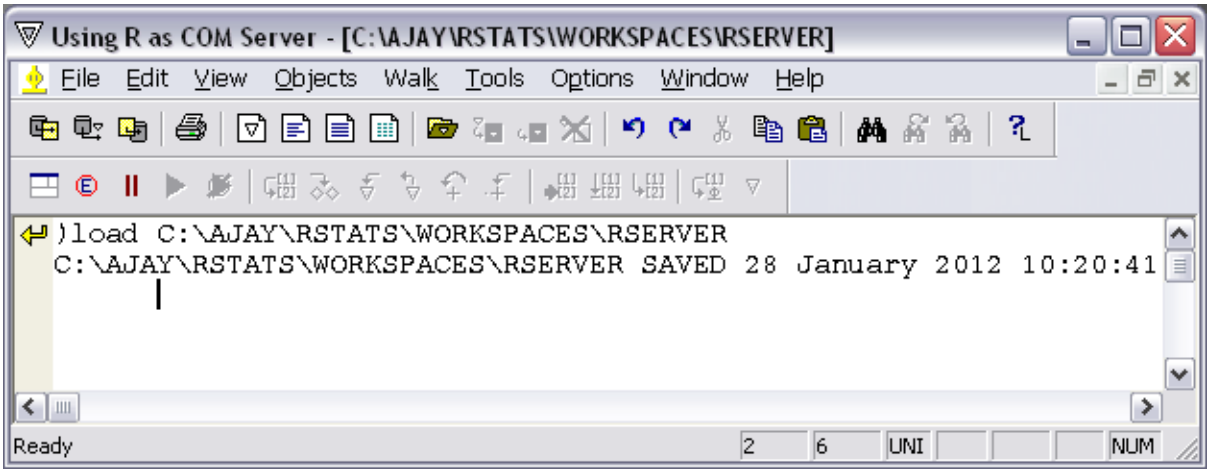
```
      ⎕wi 'XEvaluate' 'capture.output(dput(myList))'
 list(c(1, 2, 3, 4), structure(c(17, 2, 3, 4, 18, 19), .Dim = c(3L,  2L)), c("Ajay",
"Askoolum"))
```

2.3. Starting the Server

This function also sets the ⎕wi context to that instance of R. The workspace that includes this function has its latent expression set as follows:

'#' ⎕wi 'ReleaseObjects' ⋄ '#' ⎕wi 'Reset' ⋄ ⎕wcall 'SetWindowText' ('#' ⎕wi 'hwndmain') 'Using R as COM Server' ⋄ RServer

- The instance of R is created as soon as the workspace is loaded.
- The context of ⎕wself is globally set to the instance of R.
- The caption of the APL+Win window is set to 'Using R as a COM Server'



3. Using R as a Client to an APL+Win Server

If you will, pause for a moment and ponder this question: "What is the inevitable problem in using APL+Win as a server from a non-APL client?" Of course, it is the APL+Win character set: it is virtually impossible to enter expressions—which include APL+Win characters—in R for execution by APL+Win. Therefore, on the face of it, the potential for using APL+Win as a server is restricted: you can but execute function and read variables whose names do not include any APL+Win characters.

For APL+Win to act as a server that unleashes all the powers of APL, the server must recognize keywords, perhaps as an option. This might overcome most of the problem; of course, it is not available!

- I am not sure about a solution for \ which is a typewriter key and an APL primitive that denotes the start of an escape sequence in R, C#, and other languages.
- Unless the client has the facility for transposing matrices or collates elements of a matrix in the same order as APL+Win—that is, by row—this heralds another potential problem. Perhaps the APL+Win server should have a user-selectable option for specifying the collation sequence?

R has the facilities for transposing matrices and user-selectable collation sequence; however, the default is by column in common with Excel.

In order to keep matters simple, I recommend that you keep the server session visible as this allows user interaction: you will have full access, including the ability to examine objects created by the client.

Ensure that workspaces intended for loading into the server have a clear stack and do not have any graphical user interface elements: the client cannot interact with user interfaces. If you load a workspace whose stack is not clear, the client waits for the server; unless, the server is visible, this will appear as a deadlock that requires both client and server to be closed. When the server is visible, simply execute the)clear command and the client regains control.

A workspace may have a latent expression. This is handy not least because you can change the caption of the server session in order to identify it easily.

```
⎕lx←"'#' ⎕wi 'caption' 'R using APL+Win as COM Server'"
```

3.1. The 'Variable' property of an APL+Win server

Why is 'Variable' a property? Conventionally, properties have either read/write or read-only attributes. In other words, you can read or write its value. The 'Variable' property behaves like a method—it requires one argument for reading and two arguments for writing.

Although this property works fine for an APL+Win client, it does not work for some non-APL clients. With R or Excel, I have been unable to assign or create a variable in an APL+Win server instance using this *property*. A C# client manages to expose the methods *set_Variable* and *get_Variable* which, respectively, allow write and read access; these latter methods are not available with an R or Excel client. Am I missing something here? Or, is there a need for a method to enable read/write variables in an APL+Win server for non-APL clients?

I wonder whether there is a way to create the server instance in the manner that C# creates it.

3.2. APL+Win as Server

There are at least two methods for creating a server instance of APL+Win.

The first method requires no further tweaking of the R installation; this method uses rcom which is already installed.

A server instance is created as follows:

```
> aplWin<-comCreateObject("APLW.WSEngine");
> aplWin[["Visible"]]<-TRUE;
```

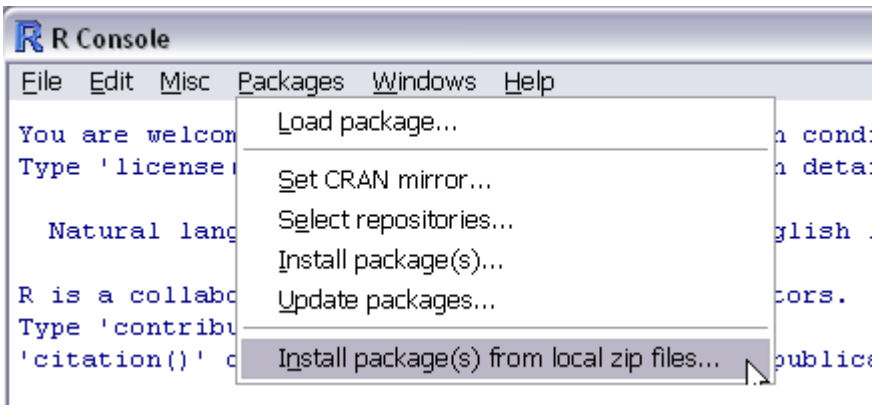
The generic recommended syntax for calling properties, methods, and events is:

comInvoke(serverInstance, property | method | event,[arg1, arg2,.. argN]).

This process automatically instantiates statconnDCOM package that is already loaded/installed with my R set-up: see the Installation section below.

The second method requires the installation of the SWinTypeLibs package. I am using the version from http://www.omegahat.org/R/bin/windows/contrib/2.14/; I am using the package SWinTypeLibs_0.6-0.zip which requires RDCOMClient_0.93-0.zip also available at the same location.

Download and save both files locally. In order to install the package, select **Packages | Install Packages from local zip files** … and then select the package: SWinTypeLibs_0.6-0.zip



The package installation process echoes the following to the console:

```
> utils:::menuInstallLocal()
package 'SWinTypeLibs' successfully unpacked and MD5 sums checked
```

A server instance is created as follows:

```
> library(SWinTypeLibs)' # This is required!
Loading required package: RDCOMClient
> apl<-COMCreate("APLW.WSEngine");
> v
```

The generic recommended syntax for calling properties, methods, and events is:

.COM(serverInstance,property | method | event, arg1, arg2,.. argN).

3.3. APL+Win with keywords

A visible APL+Win server session is also handy for constructing APL expressions. In order to use the full capability of APL+Win, my approach is to use APL+Win to render APL expressions using keywords and then pasting them in the R Client; when R calls the server, it can reverse the rendering, execute the expression and return the result.

A short demonstration will clarify this proposition. Suppose you want to evaluate the following expression from R:

```
(+\5/1)∘.×3 5.1 23 ⍝ Note: expression includes R escape character \
```

Using my scheme of naming APL primitives, this expression will become:

```
      APLToR '(+\5/1)∘.×3 5.1 23'
(+#expL#5/1)#jot#.#times#3 5.1 23
```

R can execute the expression as follows:

```
> aplWin<-comCreateObject("APLW.WSEngine");
> aplWin[["Visible"]]<-TRUE;
> aplWin$SysCommand("load C:/AJAY/RSTATS/WORKSPACES/RCLIENT");
NULL
> myMatrix<-t(comInvoke(aplWin,"Call","Expr","(+#expL#5/1)#jot#.#times#3 5.1 23"));

> myMatrix
     [,1] [,2] [,3]                                    (+\5/1)∘.×3 5.1 23
[1,]    3  5.1   23                                  3  5.1  23
[2,]    6 10.2   46                                  6 10.2  46
[3,]    9 15.3   69                                  9 15.3  69
[4,]   12 20.4   92                                 12 20.4  92
[5,]   15 25.5  115                                 15 25.5 115
```

Encouraging? I think so, definitely. How is it put together?

```
    ∇ R←Expr R;fn
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   0 0⍴⍎def ⊃'R←fn R' ('R←',RToAPL R)
[3]   R←fn R
    ∇
```

This function creates a local function that contains the expression after it has had the keywords replaced by APL+Win symbols.

```
    ∇ Z←L RToAPL R;i;j
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   (L Z)←APLToRMap R
[3]   ((⎕io+1)⊃L)←'#',¨((⎕io+1)⊃L),¨'#'
[4]   :for j :in ↑L                ⍝ i is target
[5]       i←↑(⎕io+1)⊃L ⍝ j is replacement
[6]       R←i⍷Z
[7]       :if 1∈R
[8]           Z←(~R)⊂Z
[9]           Z←((1≠⍴Z)↓⊂''),Z
[10]          R←0=2|+\(⍴Z)/1
[11]          (1↓Z)←(¯1+⍴i)↓¨1↓Z
[12]          (1↓Z)←(⊂j),¨1↓Z
[13]          Z←∊Z
[14]      :endif
[15]      L←1↓¨L
[16]  :endfor
    ∇
```

The function RToAPL translates tokens/keywords back into primitives.

The right-hand argument is a string; if you use double quotes, prefix the expression by a space.

```
    ∇ Z←L APLToR R;i;j;k
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   (L Z)←APLToRMap R
[3]   :for i :in ↑L                ⍝ i is target
[4]       j←1⌽'##',∊↑(⎕io+1)⊃L ⍝ j is replacement
[5]       R←(1⊃L)∊⊂i
[6]       :if 1∈R←i⍷Z
[7]           Z←(~R)⊂Z
[8]           Z←((1≠⍴Z)↓⊂''),Z
[9]           R←0=2|+\(⍴Z)/1
[10]          :if 1≠⍴i
```

The function APLToR translates APL symbols into keywords.

The right-hand argument is a string; if you use double quotes, prefix the expression by a space.

```
[11]              Z←(R×¯1+⍴i)↓¨Z
[12]          :endif
[13]          (¯1↓Z)←¯1↓Z,¨⊂j
[14]          Z←∊Z
[15]      :endif
[16]      L←1↓¨L
[17]  :endfor
    ∇
```

Both functions use the same primitive to keyword mapping function—this ensures consistency.

This function should be self-explanatory.

The function does not distinguish between lines that are executable and those that are comments.

It uses a primitive, keyword pair. The keywords must be unique so that there is a one to one mapping between primitive and keyword.

The commented lines have symbols which are portable between R and APL+Win. For example, the symbols < > / are not translated; therefore, if a function that contains XML or HTML strings is mapped, those elements will remain unaffected.

```
      ⎕vr 'Tags'
    ∇ Z←Tags R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   Z←⊂'<html>'
[3]   Z←Z,'<body>'
[4]   Z←Z,'<table border="1">'
[5]   Z←Z,'<tr>'
[6]   Z←Z,'<th><b>Radius</b></th>'
[7]   Z←Z,'<th><b>Area</b></th>'
[8]   Z←Z,'</tr>'
[9]   Z←Z,'<tr>'
[10]  Z←Z,'<td>',(⍕R),'</td>'
[11]  Z←Z,'<td>',(⍕○1×R⋆2),'</td>'
[12]  Z←Z,'</tr>'
[13]  Z←Z,'</table>'
[14]  Z←Z,'</body>'
[15]  Z←Z,'</html>'
    ∇

      Z←APLToR ,⎕tcnul,⎕cr 'Tags' ⋄ ⊃(Z≠⎕tcnul)⊂Z
Z#assign#Tags R
#comment# Ajay Askoolum - APL2000 Conference April 22-24, 2012
Z#assign##enclose##sq#<html>#sq#
Z#assign#Z,#sq#<body>#sq#
Z#assign#Z,#sq#<table border=#dq#1#dq#>#sq#
Z#assign#Z,#sq#<tr>#sq#
Z#assign#Z,#sq#<th><b>Radius</b></th>#sq#
Z#assign#Z,#sq#<th><b>Area</b></th>#sq#
Z#assign#Z,#sq#</tr>#sq#
Z#assign#Z,#sq#<tr>#sq#
Z#assign#Z,#sq#<td>#sq#,(#fmt#R),#sq#</td>#sq#
Z#assign#Z,#sq#<td>#sq#,(#fmt##trig#1#times#R⋆2),#sq#</td>#sq#
Z#assign#Z,#sq#</tr>#sq#
Z#assign#Z,#sq#</table>#sq#
Z#assign#Z,#sq#</body>#sq#
Z#assign#Z,#sq#</html>#sq#
```

Note: The HTML tags are unaffected.

```
      Y←RToAPL Z ⋄ ⊃(Y≠⎕tcnul)⊂Y
Z←Tags R
⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
Z←⊂'<html>'
Z←Z,'<body>'
Z←Z,'<table border="1">'
Z←Z,'<tr>'
Z←Z,'<th><b>Radius</b></th>'
Z←Z,'<th><b>Area</b></th>'
Z←Z,'</tr>'
Z←Z,'<tr>'
Z←Z,'<td>',(⍕R),'</td>'
Z←Z,'<td>',(⍕○1×R⋆2),'</td>'
Z←Z,'</tr>'
Z←Z,'</table>'
```

```
    ∇ Z←APLToRMap
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   Z←0/⊂''
[3]   ⍝ The next two are problematic; first one has 2 characters and the second, \ is \\ (escape character
[4]   Z←Z,⊂'⍝∇ pc'
[5]   Z←Z,⊂'\ expL'
[6]   ⍝ Hereon, the commented lines indicate symbols that transfer without problem
[7]   Z←Z,⊂'⎕lock'
[8]   ⍝ Z←Z,⊂'# nedt'
[9]   Z←Z,⊂'⊥ base'
[10]  ⍝ Z←Z,⊂'! bin'
[11]  ⍝ Z←Z,⊂'? deal'
[12]  Z←Z,⊂'÷ div'
[13]  Z←Z,⊂'⍟ log'
[14]  Z←Z,⊂'Ǎ divinv'
[15]  Z←Z,⊂'⌈ max'
[16]  Z←Z,⊂'⌊ min'
[17]  ⍝ Z←Z,⊂'- minus'
[18]  ⍝ Z←Z,⊂'+ plus'
[19]  ⍝ Z←Z,⊂'⋆ power'
[20]  Z←Z,⊂'⊤ rep'
[21]  Z←Z,⊂'| residue'
[22]  Z←Z,⊂'× times'
[23]  Z←Z,⊂'○ trig'
[24]  ⍝ Z←Z,⊂'∧ and'
[25]  ⍝ Z←Z,⊂'= eq'
[26]  ⍝ Z←Z,⊂'> gt'
[27]  Z←Z,⊂'≥ ge'
[28]  ⍝ Z←Z,⊂'< lt'
[29]  Z←Z,⊂'≤ le'
[30]  ⍝ Z←Z,⊂'≡ match'
[31]  Z←Z,⊂'⍲ nand'
[32]  ⍝ Z←Z,⊂'~ not'
[33]  Z←Z,⊂'≠ ne'
[34]  Z←Z,⊂'⍱ nor'
[35]  Z←Z,⊂'∨ Or'
[36]  Z←Z,⊂'⍴ shape'
[37]  Z←Z,⊂'⍒ down'
[38]  Z←Z,⊂'⍋ up'
[39]  Z←Z,⊂'⍳ index'
[40]  Z←Z,⊂'⍎ exec'
[41]  Z←Z,⊂'⍕ fmt'
[42]  Z←Z,⊂'¨ each'
[43]  ⍝ Z←Z,⊂'. dot'
[44]  Z←Z,⊂'∘ jot'
[45]  Z←Z,⊂'⍤ compF'
[46]  ⍝ Z←Z,⊂'/ compL'
[47]  Z←Z,⊂'⍣ expF'
[48]  ⍝ Z←Z,⊂', cat'
[49]  Z←Z,⊂'⍪ catAlt'
[50]  Z←Z,⊂'↓ drop'
[51]  Z←Z,⊂'⊂ enclose'
[52]  Z←Z,⊂'∈ enlist'
[53]  Z←Z,⊂'⍷ find'
[54]  Z←Z,⊂'⌷ index'
[55]  Z←Z,⊂'⊃ disclose'
[56]  Z←Z,⊂'⍥ rrf'
[57]  Z←Z,⊂'⍢ rrl'
[58]  Z←Z,⊂'↑ take'
[59]  Z←Z,⊂'⍉ transpose'
[60]  Z←Z,⊂'← assign '
```

```
Z←Z,'</body>'
Z←Z,'</html>'
```

Although, whole functions may be mapped from primitive to keyword and back again, there is no point in doing so: it would be preferable to save the function in the workspace that the server will load and have R call the function.

```
[61]  Z←Z,⊂'¯ neg'
[62]  ⍝ Z←Z,⊂'[ li'
[63]  ⍝ Z←Z,⊂'] ri'
[64]  ⍝ Z←Z,⊂'( lb'
[65]  ⍝ Z←Z,⊂') rb'
[66]  Z←Z,⊂'⍞ out'
[67]  Z←Z,⊂'⍞ prompt'
[68]  Z←Z,⊂'⍬ zilde'
[69]  Z←Z,⊂'→ branch'
[70]  Z←Z,⊂'⍝ comment'
[71]  ⍝Z←Z,⊂': label'
[72]  Z←Z,⊂'⋄ multis'
[73]  ⍝ Z←Z,⊂'; asep'
[74]  Z←Z,⊂'" sq"
[75]  Z←Z,⊂'" dq'
[76]  Z←Z,⊂'∇ del'
[77]  Z←Z,⊂'∆ delta'
[78]  Z←Z,⊂'⍞ deltau'
[79]  ⍝ Z←Z,⊂'_ uscore'
[80]  Z←(Z≠¨' ')⊂¨Z
[81]  Z←(↑¨Z)((⎕io+1)⊃¨Z)
      ∇
```

Another demonstration: this time using the second method of COM invocation.

```
      APLToR '+/1 1⍴(⍳12)∘.×⍳12'
+/1 1#transpose#(#index#12)#jot#.#times##index#12
```

```
> library(SWinTypeLibs);
Loading required package: RDCOMClient
> apl<-COMCreate("APLW.WSEngine");
> apl[["Visible"]]<-TRUE;
> .COM(apl,"SysCommand","load C:/AJAY/RSTATS/WORKSPACES/RCLIENT");
NULL
> .COM(apl,"Call","Expr","+/1 1#transpose#(#index#12)#jot#.#times##index#12");
[1] 650
```

It will take some practice—and, of course more field tests—to remember the primitive to keyword map but when that is achieved, you can edit your APL+Win functions in any text editor[13].

3.3.1. Using rcom

With the Installation as described, an R session is ready to create the APL+Win server. This is the recommended approach as it is robust and copes with APL matrices and works much more reliably than the one relying on SWinTypeLibs[14].

```
> if (!require(rcom)){
+     install.packages("rcom");
+    }
> aplWin<-comCreateObject("APLW.WSEngine");
> aplWin[["Visible"]]<-TRUE;
> comInvoke(aplWin,"SysCommand","load C:\\AJAY\\RSTATS\\WORKSPACES\\RSERVER");
NULL
> utils::setWindowTitle("R with APL+Win COM Server");This script
```

- Verifies that rcom and dependencies are available and if not, rcom is installed.
- Makes the APL+Win server visible.
- Loads a workspace in the server; the workspace has a latent expression that changes the caption of the server.
- Changes the caption of the R session.

When the workspace is loaded, the return value is NULL. Depending on the method called and with other COM servers such as Excel, the return value may be an object; everything in R is an object. However, a return value of NULL may also indicate an error condition—more about this later. The latent expression is:

[13] Incidentally, the latter demonstration was deliberately constructed so that it returned a scalar.
[14] This is unable to cope transparently with data transfer.

```
'#' ⎕wi 'caption' 'R using APL+Win as COM Server'
```

Changing the captions of the client and server is not necessary; it is just a precautionary safeguard that allows identification.

3.3.1.1. Properties

APL+Win server exposes three properties, SysVariable—for managing system variables, Variable—for managing user-defined variables in the workspace, and Visible. Note that COM interface does not recognise the X prefix and is case-sensitive.

3.3.1.1.1. SysVariable

APL+Win has a collection of system variables, such as ⎕io, ⎕rl,⎕ts etc. However, reference to these variables does not require the ⎕.

```
> comGetProperty(aplWin,"SysVariable","IO");
[1] 1
> comGetProperty(aplWin,"SysVariable","IO",0); # Should error
NULL
> !comGetProperty(aplWin,"SysVariable","IO",0); # Should error
Error in !comGetProperty(aplWin, "SysVariable", "IO", 0) :
  invalid argument type
```

The expression that returned NULL should raise an error: we are reading a property but are providing a value for it! An un-documented feature of the interface is to prefix expressions with an exclamation mark (!) to elicit a more verbose response. Use this techniques sparingly a debugging tool.

On changing index origin, the response is again NULL but this time it is not an error.

```
> comSetProperty(aplWin,"SysVariable","IO",0);
NULL
> comGetProperty(aplWin,"SysVariable","IO");
[1] 0
```

There is no means of enumerating the collection of system variables via the COM interface: see the APL+Win help files for the list.

3.3.1.1.2. Variable

Unlike system variables, the list of user—defined variables is not finite and variables can be created arbitrarily. The syntax for querying this collection is the same as with system variables.

```
> comSetProperty(aplWin,"Variable","myVal",4); # Create a variable by Value
NULL
> comGetProperty(aplWin,"Variable","myVal"); # Read a variable
[1] 4
```

When using APL+Win as the client, a variable can be created using an expression. However, although it is not immediately obvious, it is the client rather than the server that evaluates the expression. Likewise with R; however, the expression must be a valid R –not APL+Win—expression

See Exec for a means of enumerating the list of variables.

3.3.1.1.3. Visible

This property simply toggles the visibility of the server; it has two values, TRUE or FALSE.

Incidentally, unlike the second COM interface—discussed in Using SWinTypeLibs, this interface evaluates TRUE to 1 and FALSE to 0.

```
> 100 * TRUE
[1] 100
```

3.3.1.1.4. Methods

Invoking methods in the APL+Win server may or may not return a result and may or may not require one or more arguments. In order to be able to invoke APL+Win methods, some fluency with APL is a must.

3.3.1.1.5. Call

This method invokes the execution of a user-defined function in the server; that function must exist or an error occurs. Consider these functions:

```
    ∇ R←Nil
[1]   R←⎕ts
    ∇
```

```
    ∇ R←One R
[1]   R←5+R
    ∇
```

```
    ∇ R←L Two R
[1]   R←R÷L
    ∇
```

```
> a<-comInvoke(aplWin,"Call","Nil"); #Expect a vector
> a
[1] 2012    3   23   22   18   47  109
> typeof(a)
[1] "integer"

> b<-comInvoke(aplWin,"Call","One", 1);
> b
[1] 6

> comInvoke(aplWin,"Call","Two",90,1.5);# expect 90/1.5
[1] 60
```

3.3.1.1.6. Exec

This method invokes the evaluation of an APL expression in the server; the expression may involve an APL function. Unless the whole expression is valid, an error occurs.

```
> comInvoke(aplWin,"Exec","ListVariables"); # Enumerate user defined variables
[1] "dept       " "depts      " "dow        " "dowI       " "dowdowIdoww"
[6] "doww       "
> comInvoke(aplWin,"Exec","ListFunctions"); # Enumeratre user defined variables
[1] "RBServer          " "RFServer          " "RServerx          "
[4] "ReadDefinition    " "ReadList          " "ReceiveVector     "
[7] "ReceiveVectorType"
```

The two methods—note only a partial list is returned to keep the response to a minimum—are defined as follows:

```
    ∇ R←ListVariables
[1]   R←'d' ⎕nl 2
    ∇
```

```
    ∇ R←ListFunctions
[1]   R←'R' ⎕nl 3
    ∇
```

Strictly speaking, the appropriate server method to use is Call since Exec is meant to evaluate expressions. However, a function that takes no arguments is an expression.

3.3.1.1.7. SetOrphanTimeout

Refer to the like named topic in the next section.

3.3.1.1.8. SysCall

This method allows the invocation of APL+Win system functions.

```
> comInvoke(aplWin,"SysCall","wsid");
[1] "C:\\AJAY\\RSTATS\\WORKSPACES\\RSERVER"
> comInvoke(aplWin,"SysCall","wssize");
[1] 1503920128
> comInvoke(aplWin,"SysCall","sysver");
[1] "11.1.03  Jan  5 2012 14:05:58  Win/32"
```

3.3.1.1.9. SysCommand

This method allows the invocation of APL+Win system commands—these begin with) I APL but are omitted when using COM automation.

```
> comInvoke(aplWin,"SysCommand","wsid newCopy");
NULL
> comInvoke(aplWin,"SysCall","wsid");
[1] "C:\\WINDOWS\\SYSTEM32\\NEWCOPY"
```

3.3.1.2. Events

The rcom interface does not handle interfaces.

3.3.2. Advanced investigation of the Exec method

As stated, the Exec method evaluates APL expressions and returns the result to the client. With R as the client, the usefulness of this method is severely curtailed because it is impossible to enter APL+Win primitive symbols from R. During the preceding discussion of this issue, the resolution was to use APL+Win keywords. What follows is a variation on this theme.

It is quite easy to include APL+Win expressions—with symbols that will map correctly inside APL+Win—with a little bit of subterfuge. However, this is practical only in circumstances where the expressions are known in advance. With prior knowledge

- An APL+Win active session can render any APL expression with keywords.
- The converted expression can be pasted into an R script that has a call to the APL+Win server to translate the expression back into symbols.
- R can then invoke the execution of the expression with symbols.

A couple of examples might clarify the foregoing.

3.3.2.1. Example 1

Suppose the intention is to execute the following:

```
⎕rl←1903 ⋄ a←5?10 ⋄ b←3?100 ⋄ a∘.+b ⍝ Good number of primitives here!
```

Step 1: Convert the expression into keywords using APL+Win.

```
      APLToR '⎕rl←1903 ⋄ a←5?10 ⋄ b←3?100 ⋄ a∘.+b'
#out#rl#assign#1903 #multis# a#assign#5?10 #multis# b#assign#3?100 #multis# a#jot#.+b
```

Step 2: Paste the expression in R and call APL+Win to convert it back to symbols.

```
> aplKeyword<-"#out#rl#assign#1903 #multis# a#assign#5?10 #multis# b#assign#3?100 #multis#
a#jot#.+b";
> aplExp<-comInvoke(aplWin,"Call","Expr2",aplKeyword);
> aplExp;# Wow! That's how R sees the APL primitive symbols
[1] "•rl\0061903 \004 a\0065?10 \004 b\0063?100 \004 a∘.+b"
```

Step3: Invoke the Exec method and have the server evaluate the expression

```
> myMatrix<-as.matrix(t(comInvoke(aplWin,"Exec",aplExp)));# The results need transposition
```

The results are the same!

```
> myMatrix; #Browse the results
     [,1] [,2] [,3]
[1,]   43   57   22
[2,]   40   54   19
[3,]   34   48   13
[4,]   35   49   14
[5,]   41   55   20
> rowSums(myMatrix);
[1] 122 113  95  98 116
```

```
      ⎕rl←1903 ⋄ a←5?10 ⋄ b←3?100 ⋄
⎕←myMatrix←a∘.+b
 43 57 22
 40 54 19
 34 48 13
 35 49 14
 41 55 20
      +/myMatrix
122 113 95 98 116
```

3.3.2.2. Example 2

The objective is to specify the following expression from R:

```
⎕rl←1988 ⋄ R←20 50 ⋄ L←10?100 ⋄ (×R[⎕io]-L)+×R[⎕io+1]-L
```

Step 1: Convert the expression into keywords using APL+Win[15].

```
      APLToR "⎕rl←1988 ⋄ R←20 50 ⋄ L←10?100 ⋄ (×R[⎕io]-L)+×R[⎕io+1]-L"
#out#rl#assign#1988   #multis#   R#assign#20   50   #multis#   L#assign#10?100   #multis#
(#times#R[#out#io]-L)+#times#R[#out#io+1]-L
```

[15] The result is a vector, albeit it appears on two physical lines.

Step 2: Paste the expression in R and call APL+Win to convert it back to symbols.

```
> aplExp<-comInvoke(aplWin,"Call","Expr2",aplKeyword);aplKeyword<-" #out#rl#assign#1988
#multis# R#assign#20 50 #multis# L#assign#10?100 #multis# (#times#R[#out#io]-
L)+#times#R[#out#io+1]-L ";
> aplExp<-comInvoke(aplWin,"Call","Expr2",aplKeyword);
> aplExp; #See what it looks like
[1] " •rl\0061988 \004 R\00620 50 \004 L\00610?100 \004 (’R[•io]-L)+’R[•io+1]-L "
```

Step3: Invoke the Exec method and have the server evaluate the expression

```
> comInvoke(aplWin,"Exec",aplExp);
 [1] -2 -1 -2  1 -2 -2  0  0 -2 -2
```

```
      ⎕rl←1988 ⋄ R←20 50 ⋄ L←10?100 ⋄ (×R[⎕io]-
L)+×R[⎕io+1]-L
¯2 ¯1 ¯2 1 ¯2 ¯2 0 0 ¯2 ¯2
```

Note the transparent translation of APL negative numbers.

In case this needs to be pointed out, setting the APL+Win random link ensures that APL roll returns the same numbers.

How is Expr2 defined?

```
    ∇ R←Expr2 R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   R←RToAPL R
    ∇
```

3.3.2.3. Example 3

Of course with some practice converting the APL+Win primitive to its keyword mapping, it becomes possible to eliminate step one altogether: you can type the expression directly in R. This creates much more interesting possibilities.

Which elements on the diagonal of `(⍳12)∘.×(⍳12)` are odd? The APL+Win expression that yields the answer is:

```
      (2|a)/a←1 1⍴⍨(⍳12)∘.×⍳12 ⍝ Always work in ⎕io←1 with R
1 9 25 49 81 121
```

```
> # Create the transliterated expression directly in R
> # Use a reference card showing the primitive/keyword map, if necessary.
> aplExp<-"(2#residue#a)/a#assign#1 1#transpose#(#index#12)#jot#.#times##index#12"

> aplExpMapped<-comInvoke(aplWin,"Call","Expr2",aplExp);
> aplExpMapped; # What does it look like?
[1] "(2þa)/a\0061 1⍳(â12)ø.’â12"
> comInvoke(aplWin,"Exec",aplExpMapped); # Evaluate it
[1]   1   9  25  49  81 121
```

3.3.3. Using SWinTypeLibs

The documentation[16] and material you can find on the internet on this option for interfacing R and other COM servers tend to concentrate on Excel and to a lesser extent on Word. The worked examples certainly work and they illustrate a means of including R output in documents.

However, with APL+Win, this route to automation is fraught with difficulties and I do not recommend it for one important reason: the interface is not transparent. It is possible that a later revision of this package will overcome the issues but for now:

- There appears to be issues with the transfer of some data types[17].
- Data structures do not transfer: it is not possible to receive matrices from the APL+Win server unless the server coerces the result that it passes.

The dynamics of client/server automation works only when data transfers are transparent and data coercion, where necessary, are silent; in this case they are not.

Consider an example: the objective is to retrieve an array—for definition, see code below—from APL+Win.

[16] See http://www.baselr.org/Manipulating%20Office%20Documents%20with%20R%20-%20Jan%202012.pdf.
[17] See http://www.stat.berkeley.edu/~nolan/stat133/Fall05/lectures/DCOM.html for the list of data types supported.

```
    ∇ Z←RR
[1]   ⎕rl←1924
[2]   Z←2 3 5⍴(0.001×10?1000)×10?123213
[3]   Z←'#' ⎕wi 'VT' (Z) 8204
    ∇

      RR
     VT 8204      17354.194 72990.145  5917.305 33087.372 16487.52
                  75934.913 42228.795 20019.368  1271.124 31434.48
                  17354.194 72990.145  5917.305 33087.372 16487.52

                  75934.913 42228.795 20019.368  1271.124 31434.48
                  17354.194 72990.145  5917.305 33087.372 16487.52
                  75934.913 42228.795 20019.368  1271.124 31434.48
```

```
> library(SWinTypeLibs);
> apl<-COMCreate("APLW.WSEngine");
> apl[["Visible"]]<-TRUE;
> apl$SysCommand("load C:/AJAY/RSTATS/WORKSPACES/COM4R");
NULL
> a<-.COM(apl,"Call","RR");
> typeof(a); # It is not an array!
[1] "list"
> dput(a);   # It is a list
list(list(list(95774.539, 1507.113, 118665.678, 93102.093, 24878.952),
    list(16098.62, 7460.772, 28692.185, 48949.74, 4621.449),
    list(95774.539, 1507.113, 118665.678, 93102.093, 24878.952)),
    list(list(16098.62, 7460.772, 28692.185, 48949.74, 4621.449),
        list(95774.539, 1507.113, 118665.678, 93102.093, 24878.952),
        list(16098.62, 7460.772, 28692.185, 48949.74, 4621.449)))
> b<-as.array(.COM(apl,"Call","RR")); # Coerce
> typeof(b); # Still a list!
[1] "list"
```

Even with data coercion in R—note the as.array coercion—the received data is a list rather than an array.

Without data coercion within APL+Win—see line [3] of the APL function—the R session locks up. Therefore, the investigation of this route is fairly basic: I will restrict examples to scalars. There is some documentation on this approach at this location: http://www.stat.berkeley.edu/~nolan/stat133/Fall05/lectures/DCOM.html. At the time of writing, it appears incomplete or unfinished. However, it does elaborate on strategies for handling events and exposing the collection of properties, methods, and events using Excel as the server.

Note that the COM interface is indifferent as regards the case when property, method, or event names are specified: all are specified as literals.

The package dependencies are as follows:

SWinTypeLibs	Reads meta-information from COM objects; it requires RDOMClient.
RDCOMClient	Enables COM from R.
RDCOMServer	Exports R functions to other applications; I am not using this package since it depends on yet more packages and it changes the Registry. For details, consult http://www.omegahat.org/RDCOMServer/. And for examples, see http://www.omegahat.org/RDCOMServer/examples/.

This interface uses $ where dot (.) is used with object oriented instances of objects. Therefore, the following two lines are equivalent:

```
> apl$Visible(); # Read Only.Note () to indicate that the function is called
[1] -1
> apl[["Visible"]]; # Read Write
[1] -1
```

Of course, in the realm of R, there are other pathways!

```
> .COM(apl,"Visible"); # Read Only
[1] -1
```

3.3.3.1. Properties

Properties or attributes—in plain terms, variables and constants—are either read/write or read-only. The conventional syntax of this COM interface will allow the R client to query or assign a property.

With this interface, a property that holds a value, such as the Visible property, both read and write are supported. With a property that holds a collection, such as the SysVariable property, only read access is permitted.

```
> apl$SysVariable("io"); # Read Access
[1] 1
> apl$SysVariable("io")<-0;
Error in apl$SysVariable("io") <- 0 :
  target of assignment expands to non-language object
> .COM(apl,"SysVariable","io");
[1] 1
> .COM(apl,"SysVariable","io",0);
Error: Invalid number of parameters.
```

3.3.3.1.1. SysVariable

This property holds a collection of APL System variables e.g. IO,WA,WSSIZE etc.; they can be read only, although they are read/write properties.

```
> .COM(apl,"SysVariable","rl"); # Read only
[1] 16807
```

3.3.3.1.2. Variable

This property holds a collection—the set of user defined variables in the APL+Win server workspace. The same restrictions apply.

```
> .COM(apl,"Variable","ABC");
[1] 90
> apl$Variable("ABC");
[1] 90
```

3.3.3.1.3. Visible

This property is the only one that behaves as required.

```
> #Write
> apl[["Visible"]]<-TRUE;
> #Read
> apl[["Visible"]]; # Syntax 1
[1] -1
> apl$Visible();    # Syntax 2
[1] -1
> # Note it returned -1 rather than 1 … reminiscent of Visual Basic 6.0 or Excel VBA!
> .COM(apl,"Visible"); # Syntax 2
[1] -1
```

3.3.3.2. Methods

The collection of methods has dedicated methods for dealing with user defined and system methods. For user-defined methods, that is functions, use Call and Exec. For system functions, use SysCall and SysCommand.

3.3.3.2.1. Call

This method allows you to call user-defined functions available in the server session. APL+Win functions can take none, one, or two arguments; a single argument is always referred to as the right-hand argument. In calling a function, the right-hand argument is specified first. Function calls match the arguments by position only: there is no other option.

Consider these examples:

```
    ∇ R←Nil              ∇ R←One R              ∇ R←L Two R              ∇ NoRes
[1]   R←⍵90          [1]   R←⍵R           [1]   R←L⍵R           [1]   x←8+9
    ∇                    ∇                      ∇                      ∇
> .COM(apl,"Call","Nil"); # Care! Function name is case-sentitive
[1] 4.49981
> .COM(apl,"Call","One",90);
[1] 4.49981
> .COM(apl,"Call","Two",90,2); # 2 is the left-hand argument
[1] 6.491853
```

The results may be assigned to R variables.

```
> logBase2<-.COM(apl,"Call","Two",90,2); # Care! Function name is case-sentitive
> logBase2;
[1] 6.491853
```

If you assign the results of an APL function that returns no result, the interface assigns NULL to the variable; this is probably more acceptable than an error being raised.

```
> a<-logBase2<-.COM(apl,"Call","NoRes"); # Care! Function name is case-sentitive
> a;
NULL
```

3.3.3.2.2. Exec

The Exec method permits the execution of arbitrary but valid APL expressions, irrespective of whether that expression returns a result. The expression may be a function; however, if it is a function that requires arguments, none can be specified. Use Call instead.

```
> .COM(apl,"Exec","3+2");
[1] 5
> .COM(apl,"Exec","NoRes");
NULL
```

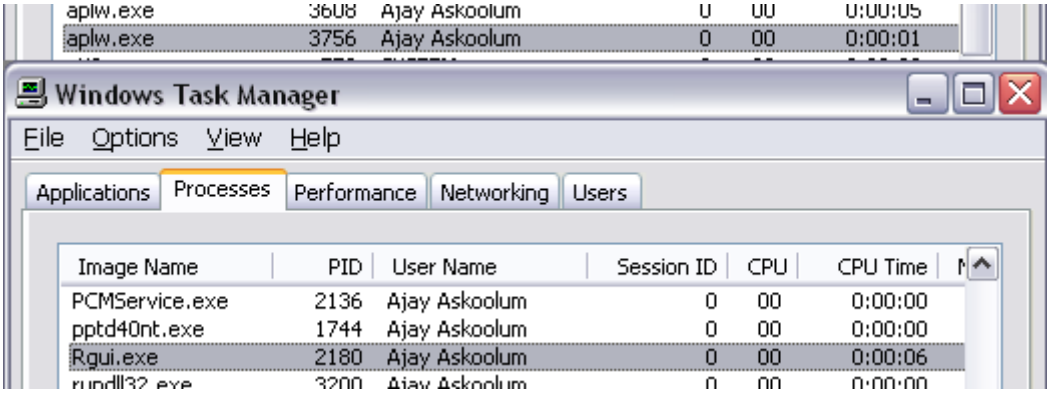
3.3.3.2.3. SetOrphanTimeout

Set orphaned-process timeout and parent process id; returns child process id.

```
> .COM(apl,"SetOrphanTimeout"); # Read
```

I expected the following result since R is the child process:

```
> .COM(apl,"SetOrphanTimeout",10,2180); # Write
[1] 3756
```



Windows Task Manager confirms the parameter and result.

3.3.3.2.4. SysCall

This method allows the client to execute any system function in the APL+Win Server. A system function is one that begins with:

```
⍞ ⍝ For example ⎕io
```

However, the APL symbol is *not* specified and the function name is not case-sensitive.

```
> .COM(apl,"SysCall","io"); # Index Origin
[1] 1
> .COM(apl,"SysCall","rl"); # Random Lonk
[1] 500782188
```

3.3.3.2.5. SysCommand

This method allows the client to execute any system function in the APL+Win Server. A system function is one that begins with:

```
) ⍝ For example )load
```

3.3.3.3. Events

The APL+Win server has two events, namely, Notify and SysNotify. Refer to http://www.stat.berkeley.edu/~nolan/stat133/Fall05/lectures/DCOM.html for details on working with events.

4. APL+Win GUI with R plots

One of the most tangible benefits of using R as a server is the option of showing R plots on APL+Win graphical user interfaces. R can display complex data graphically. Unfortunately, the graphical displays can be just as complex unless you learn to interpret them. Consider two examples: the data for both examples are included in the basic R installation.

4.1. Demonstration 1

The first example shows a fairly basic graph that is easily understood.

APL+Win can retrieve the data:

```
      ⎕wi 'XEvaluateNoReturn' 'data(presidents);'
```

R can now plot the data, shown on the right, and save it to a file.

```
> bmp(filename = "c:/ajay/rstats/plots/presidents.bmp",width
= 600, height = 400, units = "px", pointsize = 12, bg =
"white");
> data(presidents);
> plot(presidents,main=" Quarterly Approval Ratings of US
Presidents");
> dev.off();
```

As APL+Win can show bitmap files in its 'Picture' control, in this instance the plot is written as a bitmap; several other options are available,

If the file exists, R will overwrite it silently.

The dimensions of the bitmap are chosen to match the preferred dimensions for the 'Picture' control in APL+Win.

Although I have executed the R code in R, it is equally possible to execute them from APL+Win using the XEvaluateNoReturn method.

Note that the output file is: c:/ajay/rstats/plots/presidents.bmp.

The underlying data structure is time series.

```
> presidents;
     Qtr1 Qtr2 Qtr3 Qtr4
1945   NA   87   82   75
1946   63   50   43   32
1947   35   60   54   55
1948   36   39   NA   NA
1949   69   57   57   51
1950   45   37   46   39
1951   36   24   32   23
1952   25   32   NA   32
1953   59   74   75   60
1954   71   61   71   57
1955   71   68   79   73
1956   76   71   67   75
1957   79   62   63   57
1958   60   49   48   52
1959   57   62   61   66
1960   71   62   61   57
1961   72   83   71   78
1962   79   71   62   74
1963   76   64   62   57
1964   80   73   69   69
1965   71   64   69   62
1966   63   46   56   44
1967   44   52   38   46
1968   36   49   35   44
1969   59   65   65   56
1970   66   53   61   52
1971   51   48   54   49
1972   49   61   NA   NA
1973   68   44   40   27
1974   28   25   24   24
```

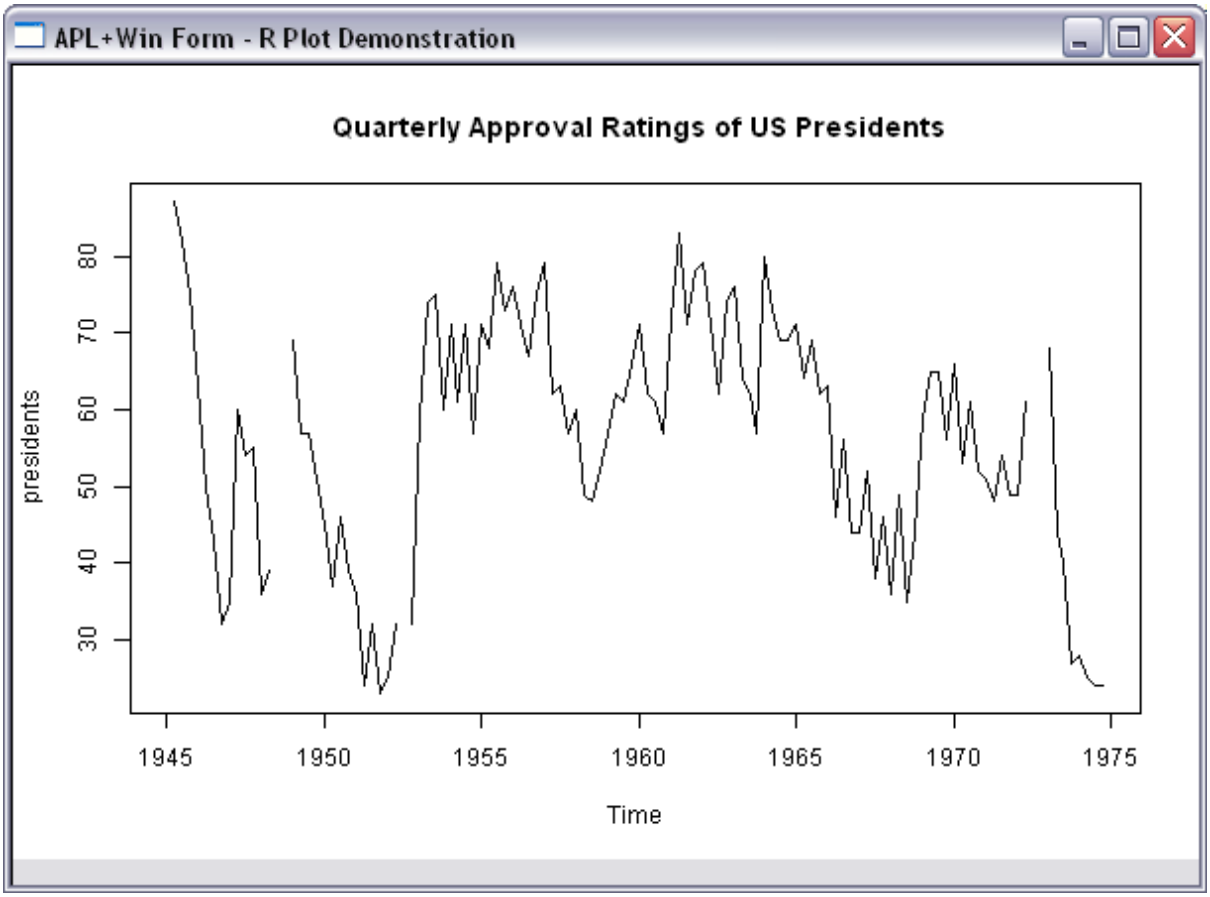
The plot instruction is basic: none of the visual attributes that are possible, for example, type of plot, colours etc. are specified.

The APL+Win function that will show the plot is:

```
    ∇ ShowPlot R;⎕wself
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕wself←'frmR' ⎕wi 'Create' 'Form'
[3]   ⎕wi 'caption' 'APL+Win Form - R Plot Demonstration'
[4]   ⎕wi 'enabled' 1
[5]   ⎕wi 'where' 16 40 28 76
[6]
[7]   ⎕wself←'frmR.im' ⎕wi 'New' 'Picture' ('scale' 5)
[8]   ⎕wi 'origin' 50 50
[9]   ⎕wi 'bitmap' R
[10]  ⎕wi 'imagesize' 400 600
[11]  ⎕wi 'extent' ('frmR' ⎕wi 'extent')
[12]  ⎕wi 'where' 0 0
[13]
[14]  'frmR' ⎕wi 'Wait'
    ∇

ShowPlot"c:\ajay\rstats\plots\presidents.bmp" ⍝ Backslash in APL+Win
```

The plot shows as follows:



4.2. Demonstration 2

The second example is slightly more complex.

This is based on a data.frame data structure.

```
> class(OrchardSprays);
[1] "data.frame"
```

This time, the plot is constructed by calling the R commands from APL+Win:

```
    ∇ OrchardSprays
[1]   ⍝ Ajay Askoolum - APL2000 Conference
April 22-24, 2012
[2]   ⎕wi 'XEvaluateNoReturn' 'bmp(filename =
"c:/ajay/rstats/plots/orchardsprays.bmp",width
= 600, height = 400, units = "px", pointsize
      = 12, bg = "white");'
[3]   ⎕wi 'EvaluateNoReturn'
'data(OrchardSprays);'
[4]   ⎕wi 'XEvaluateNoReturn'
'plot(OrchardSprays,main=" Potency of Orchard
Sprays");'
[5]   ⎕wi 'XEvaluateNoReturn' 'dev.off();'
    ∇
```

Warning: Note that I have used `⎕wi` without a left-hand argument. You will need to ensure that the left-hand argument defaults to the background or foreground instance of the R server.

Since the APL+Win form definition also assigns to `⎕wself`, the danger that some confusion is introduced does exist.

Note that both the R and APL+Win code have

```
⎕wi 'XEvaluateNoReturn'
'data(OrchardSprays);'
⊃⎕wi 'XEvaluate'
'capture.output(OrchardSprays)'

   decrease rowpos colpos treatment
1        57      1      1         D
2        95      2      1         E
3         8      3      1         B
4        69      4      1         H
5        92      5      1         G
6        90      6      1         F
7        15      7      1         C
8         2      8      1         A
9        84      1      2         C
10        6      2      2         B
11      127      3      2         H
12       36      4      2         D
13       51      5      2         E
14        2      6      2         A
15       69      7      2         F
16       71      8      2         G
17       87      1      3         F
18       72      2      3         H
19        5      3      3         A
20       39      4      3         E
21       22      5      3         D
22       16      6      3         C
23       72      7      3         G
24        4      8      3         B
25      130      1      4         H
26        4      2      4         A
27      114      3      4         E
28        9      4      4         C
29       20      5      4         F
30       24      6      4         G
31       10      7      4         B
```

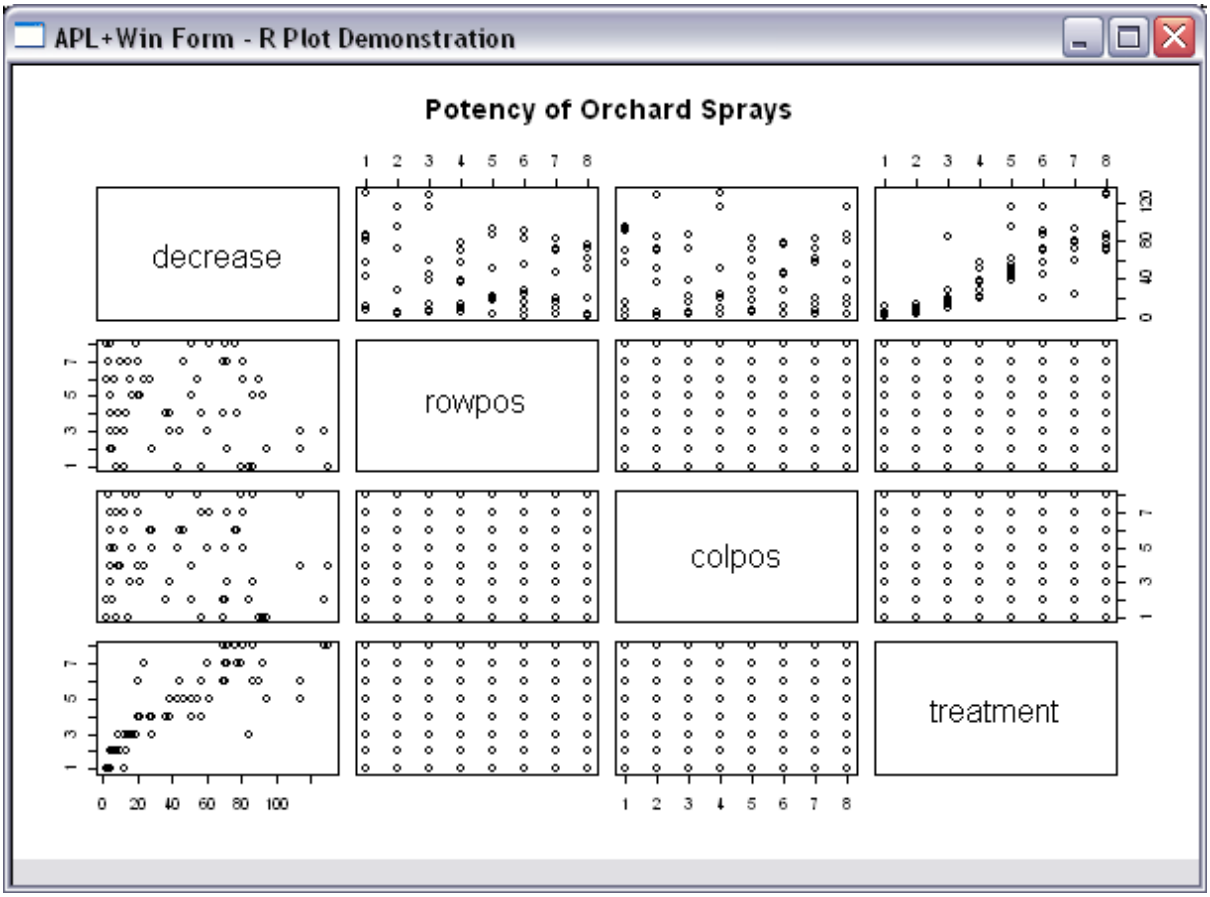
specification for width and height in pixels: make adjustments in both places so that the plots show correctly in APL+Win.

Line [8] of ShowPlot puts the image on the APL+Win form using the 'bitmap' property. The 'image' property has a method 'Paste'—from the clipboard—but it accepts a bitmap image only and I have not found a way to put a bitmap on the clipboard from R[18].

```
32  51  8  4  D
33  43  1  5  E
34  28  2  5  D
35  60  3  5  G
36   5  4  5  A
37  17  5  5  C
38   7  6  5  B
39  81  7  5  H
40  71  8  5  F
41  12  1  6  A
42  29  2  6  C
43  44  3  6  F
44  77  4  6  G
45   4  5  6  B
46  27  6  6  D
47  47  7  6  E
48  76  8  6  H
49   8  1  7  B
50  72  2  7  G
51  13  3  7  C
52  57  4  7  F
53   4  5  7  A
54  81  6  7  H
55  20  7  7  D
56  61  8  7  E
57  80  1  8  G
58 114  2  8  F
59  39  3  8  D
60  14  4  8  B
61  86  5  8  H
62  55  6  8  E
63   3  7  8  A
64  19  8  8  C
```

The plot shown on an APL+Win form:

```
ShowPlot 'c:\ajay\rstats\plots\orchardsprays.bmp'
```



[18] It can put a Windows meta-file on the clipboard.

This is a default plot; the visual representation is quite complex unless you know how to read it. In practice, researchers would try out several plots before deciding on the one suitable for the target audience.

4.3. Demonstration 3

The ability to put R plots on an APL+Win graphical interface no doubt enhances the APL+Win application. The fact that the bitmaps that APL+Win shows are stored as files also means that such plots can be included in formal reports, constructed in, say, Word documents. Remember that in the previous two demonstrations, R is the server and APL+Win the client. R can put plots directly into Word documents using COM; certainly, this is one option.

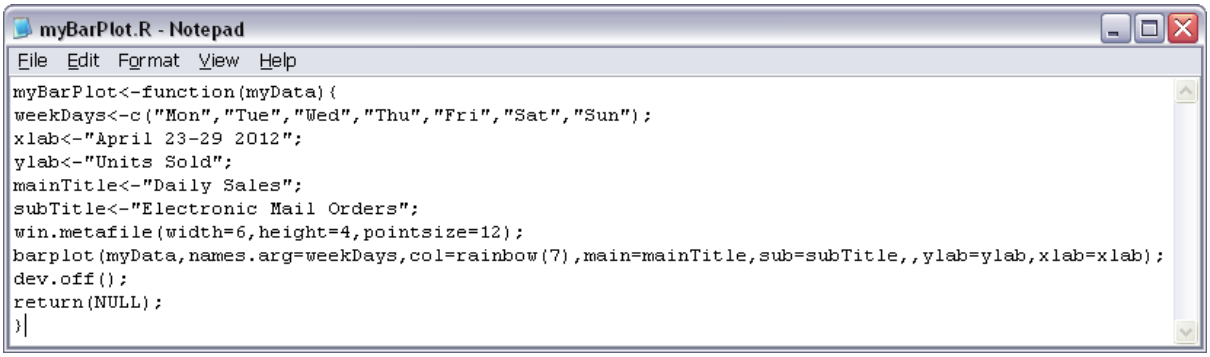
However, for APL+Win, it is almost certain that any report created in Word will contain the bitmap and other information available from APL+Win. Let me demonstrate another option: APL+Win preparing a very basic report in Word comprising of one R plot and some data from the APL+Win workspace.

```
    ∇ PlotToClipboard;⎕wself;PlotToClipboard;weekDays;xlab;ylab;mainTitle;subTitle
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕wself←'⍙rfs'
[3]   ⎕rl←1989 ⍝ So you can rereate the same data
[4]   ⍝ Create the data and attributes for the plot
[5]   myData←7?1000
[6]   weekDays←"Mon" "Tue" "Wed" "Thu" "Fri" "Sat" "Sun"
[7]   xlab←"April 23-29 2012"
[8]   ylab←"Units Sold"
[9]   mainTitle←"Daily Sales"
[10]  subTitle←"Electronic Mail Orders"
[11]  ⍝ Send plot data to R
[12]  ⎕wi 'XSetSymbol' 'myData' myData
[13]  ⎕wi 'XSetSymbol' 'weekDays' weekDays
[14]  ⎕wi 'XSetSymbol' 'xlab' xlab ⍝ x-axis label
[15]  ⎕wi 'XSetSymbol' 'ylab' ylab ⍝ y-axis label
[16]  ⎕wi 'XSetSymbol' 'mainTitle' mainTitle
[17]  ⎕wi 'XSetSymbol' 'subTitle' subTitle
[18]  ⍝ Now set the clipboard as the graphics device
[19]  ⎕wi 'XEvaluateNoReturn' 'win.metafile(width=6,height=4,pointsize=12);'
[20]  ⍝⎕wi 'XEvaluateNoReturn' 'jpeg(width = 600, height = 400, units = "px", pointsize =
12, bg = "white");'
[21]  ⍝ Plot & save to clipboard & close the device
[22]  ⎕wi 'XEvaluateNoReturn'
'barplot(myData,names.arg=weekDays,col=rainbow(7),main=mainTitle,sub=subTitle,,ylab=ylab,x
lab=xlab);'
[23]  ⎕wi 'XEvaluateNoReturn' 'dev.off();'
[24]  ⍝ Plot must be ready for pasting
    ∇
```

This function uses R to create a simple plot on the clipboard. All the data used in the plot are created within APL+Win and passed to R.

Each step in executed incrementally and in a predefined sequence. With an adequate level of fluency in R, the alternative is to write all the R code to a script file and then simply call on R to execute that script.

```
> source("drive:/path/filename.R");
```



I have saved the same plot as R code, and as a function. Now I can re-use the code any number of times from APL+Win for plotting different data—I just need to supply the data into R:

```
    ∇ PlotReUse
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕wi 'XEvaluateNoReturn' 'Source("c:/ajay/rstats/scripts/myBarPlot.R");'
[3]   ⎕wi 'XSetSymbol' myData (7?1000)
```

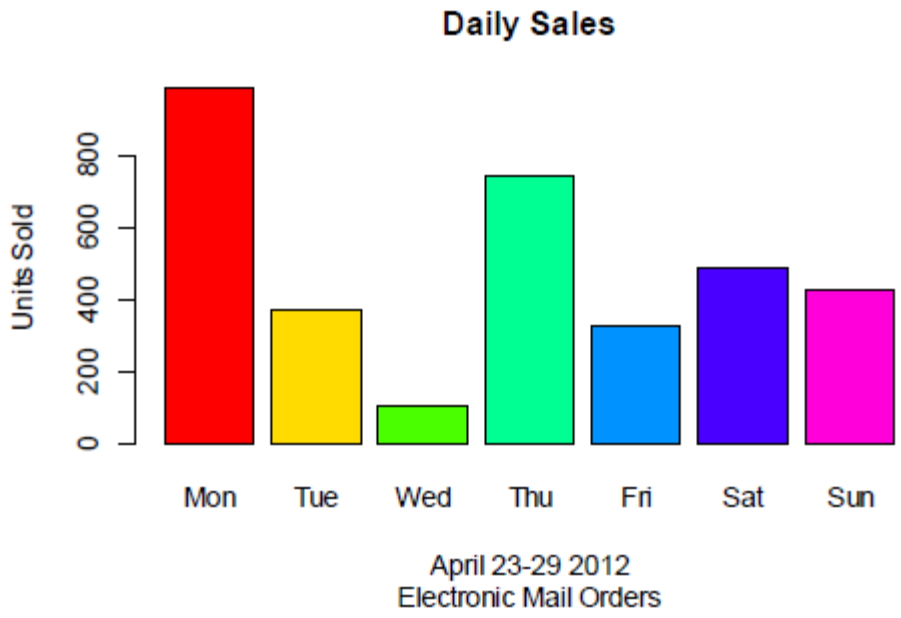
```
[4]    ⎕wi 'XEvaluateNoReturn' 'myBarPlot(myData);'
    ∇
```

On line [19], the measurement units for width and height are in inches. Line [20], which is commented out, shows the code for creating jpeg files.

Using the clipboard for holding the plot makes for a speedier alternative to writing it to a file. However, if you have a series of plots, you need to 'consume' the plot as soon as created otherwise each subsequent one overwrites the clipboard without warning

In order to understand the complexity of the plots that R can produce and indeed to learn how to code them, refer to existing examples and adapt them. This link provides an exhaustive reference with code: http://addictedtor.free.fr/graphiques/allgraph.php.

A basic APL+Win report created by APL+Win.



Based on the following data:

Mon	Tue	Wed	Thu	Fri	Sat	Sun
985	371	101	743	325	488	426

The code that created the content of the Word document is:

```
    ∇ WordReport;⎕wself;i;j
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   ⎕wself←'wd' ⎕wi 'Create' 'Word.Application'
[3]   ⎕wi 'visible' 0 ⍝ Leave it hidden for now
[4]   0 0⍴⎕wi 'XDocuments.Add'
[5]   ⎕wi 'xSelection.TypeText' 'A basic APL+Win report created by APL+Win.'
[6]   ⎕wi 'xSelection.TypeParagraph'
[7]   Plot ⍝ Create a plot in R and save it to the clipboard
[8]   ⎕wi 'xSelection.Paste'
[9]   ⎕wi 'xSelection.TypeParagraph'
[10]  ⎕wi 'xSelection.TypeText' 'Based on the following data:'
[11]  ⎕wi 'xSelection.TypeParagraph'
[12]  ⍝ 3 Rows(WeekDays, Data) 7 columns (MON ... SUN)
[13]  ⎕wi 'xActiveDocument.Tables.Add(Range,NumRows,NumColumns)' (('wd' ⎕wi
'Selection.Range')⎕wi 'obj') 2 7
[14]  :for i :in weekDays
[15]      0 0⍴⎕wi 'xSelection.TypeText' i
[16]      ⎕wi 'xSelection.MoveRight' (⎕wi '=wdCell')
[17]  :endfor
[18]  j←1
[19]  :for i :in myData
[20]      ⎕wi 'xSelection.TypeText' (⍕i)
[21]      j←j+1
```

```
[22]      :if j≤⍴myData ⍝ No tab in final cell or it will create another row
[23]          0 0⍴⎕wi 'xSelection.MoveRight' (⎕wi '=wdCell')
[24]      :endif
[25]  :endfor
[26]  ⎕wi 'xSelection.MoveDown' (⎕wi '=wdLine')1
[27]  ⎕wi 'ActiveDocument.Paragraphs().Range.Select' 1
[28]  ⎕wi 'xSelection.ParagraphFormat.Alignment' (⎕wi '=wdAlignParagraphCenter')
[29]  ⎕wi 'xSelection.Font.Bold' 1
[30]  ⎕wi 'xSelection.Font.Underline' (⎕wi '=wdUnderlineSingle')
[31]  ⎕wi 'xSelection.EndKey' (⎕wi '=wdStory')
[32]  ⎕wi 'visible' 1
    ∇
```

This function builds the report step-by-step; this is acceptable for a demonstration. In practice, for a working report, you will need to create and format the report using Word itself. The document should contain bookmarks as placeholders for the content that APL+Win will fill-in.

5. Saving/Loading R client and server objects

The R client and/or server session accumulates a set of objects—functions, homogeneous and heterogeneous data structures—during interaction with APL+Win. Unless this interaction is experimental, it is advisable to save the active R session so that it can be re-loaded during another COM session.

If you are using the foreground server, use either the R or APL+Win code for saving and loading R workspaces or objects. If you use the APL+Win code, note that the interactive dialogues do not always appear as the top window; therefore, it is advisable to use the syntax that does not require interaction.

Objects are saved to and loaded from files. Remember that \ is a reserved character in R—it denotes the start of an escape sequence; therefore, either use / or \\ instead of \. The penalty for any oversight in specifying a fully qualified file name is liable to be a frozen APL+Win session. In order to counteract this, it might be preferable to write cover functions which automatically do this substitution.

5.1. As a workspace

You can save the whole workspace to a named file as follows:

```
> save.image(file="myRWS.RDATA",ascii=FALSE); # Binary format
```

Or, use the client expression when using R as a background server:

```
⎕wi 'XEvaluateNoReturn' 'save.image(file="myRWS.RDATA",ascii=FALSE);'
```

Alternatively, use the following command from the server:

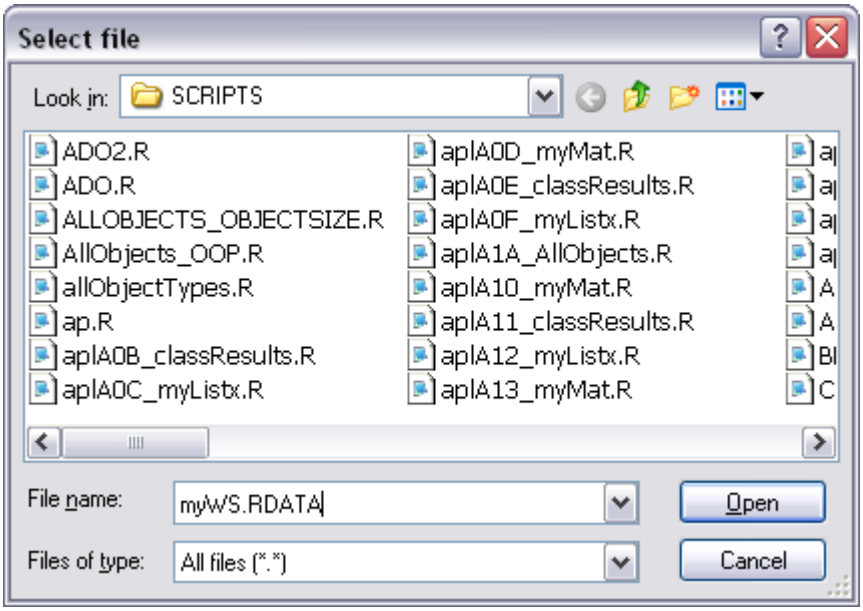
```
> save.image(file=file.choose(new=TRUE),ascii=FALSE); # Binary format
```

This allows you to specify a file name at run time; specify a file name as appropriate. If an existing file name is specified, R will overwrite it without warning.

This will prompt you for a file name. You can change the location of the file by specifying a fully qualified file name or by changing the folder visible in the 'Look in' box.

Should you click Cancel, the following self-explanatory error is raised:

```
Error in file.choose(new
= TRUE) : file choice
cancelled
```

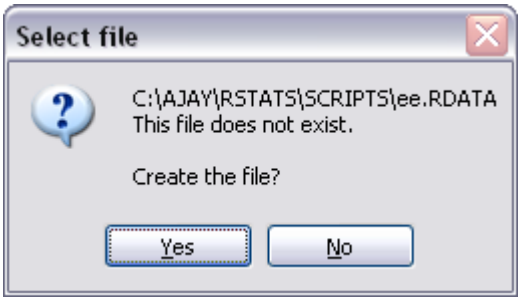


Note:

- Specifying new = FALSE allows you to select an existing file.
- Specifying ascii = TRUE allows you to save the file in text form; however, the file is NOT saved in human readable form.

You must specify a new file name with extension '.RDATA' and click Open.

Should you click No from the dialogue that follows, the prompt reverts to the original dialogue; if you click Yes, the workspace is saved in binary format.



A saved workspace may be re-loaded with the following command:

```
> load(file=file.choose(new=FALSE));
```

The equivalent APL+Win expression is:

```
      ⎕wi 'XEvaluateNoReturn' 'load(file="myfile.RDATA");'
```

Note that you need to specify the name of the file rather than choose the file dynamically:

- The *file.choose()* dialogue may not display as the top window, especially if you are using R as a background server.
- This syntax provides you with an audit trail; in other words, the code tells you what file you have loaded.

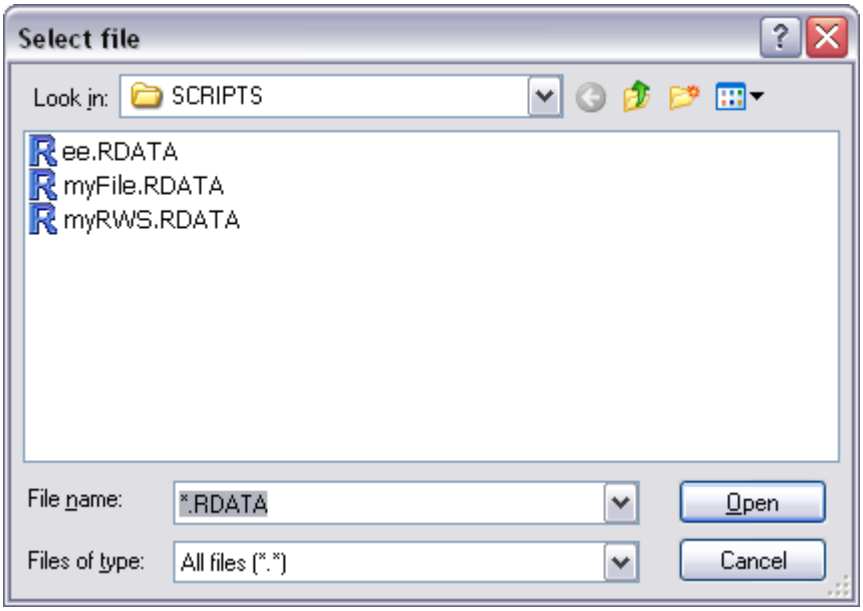
In interactive mode, you can type *.RDATA in the File Name box and click Open to display workspace file names only; this will make file selection easier.

Since it is impossible to load a file unless it exists already, set the parameter new to FALSE.

Select a workspace and click Open: the file will be loaded silently; in other words, unlike APL+Win which tells you when the file was saved, there is no response from R.

Warning: Unlike APL+Win, loading an R workspace does NOT expunge the contents of the active workspace first.

Therefore, the 'load' action is like the APL+Win 'copy' action.



If you want to clear the active R workspace before loading another, use the following command:

```
> rm(list=objects()); # Everything is an object!
```

Or, use the equivalent APL+Win expression:

```
      ⎕wi 'XEvaluateNoReturn' 'rm(list=objects())'
```

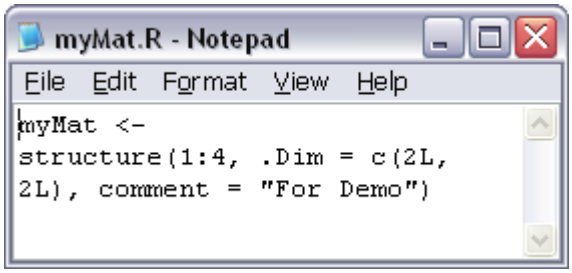
5.2. As a script

Script files offer a number of advantages.

- They are text files; therefore, they can be viewed using any text editor such as NOTEPAD.EXE.
- A script file may contain single or multiple objects, as required.
- The makeup of the active server workspace can be assembled from one or several script files.
- From the point of view of exploring R, script files allow you to see the internal definition of objects; familiarity with the alternative representations of the same object enhances fluency.
- When an object is saved as a script, its values and all its attributes—including those that assume a default value—are included.

Consider this example:

```
> myMat=matrix(c(1:4),ncol=2);
> comment(myMat)<-"For Demo";
> myMat
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> str(myMat);
 int [1:2, 1:2] 1 2 3 4
 - attr(*, "comment")= chr "For Demo"
> dput(myMat);
structure(1:4, .Dim = c(2L, 2L), comment = "For
Demo")
> dump("myMat",file="c:/myMat.R");
# The file is shown on the right.
```



On the down side, unlike workspaces, script files do not recreate the server active session exactly as where you left off and you can create clutter—multiple copies of objects in several files.

In order to save the active session as a script file, consider the following function:

```
    ∇ Z←WriteAllDefinitions
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   Z←2⊃⎕wcall 'GetTempFileName' (⎕wi 'XEvaluate' 'getwd()') "apl" 0 (256⍴⎕tcnul)
[3]   ((Z="/")/Z)←'\'
[4]   Z←((-⎕io)+L⍳⎕tcnul)↑Z
[5]   0 0⍴⎕wcall 'DeleteFile' Z
```

```
[6]   Z←(∧\'.'≠Z)/Z
[7]   ((Z="\")/Z)←'/'
[8]   Z←(Z,'_ALL#.R') InLineExpression 14 0⍕(1000,5/100)⊥6↑⎕ts
[9]   :if 0≠⎕wi 'XEvaluate' 'length(objects());'
[10]      Z←Z (⎕wi 'XEvaluate' ('dump(objects#,file="",append=TRUE);' InLineExpression
'()' Z))
[11]      (("/"=⎕io⊃Z)/⎕io⊃Z)←'\'
[12]  :else
[13]      Z←2/⊂''
[14]  :endif
[15]  ⍝ Z = fileName (list of objects written)
    ∇
      WriteAllDefinitions
 C:\AJAY\RSTATS\SCRIPTS\apl2C_ALL20120320082113.R  AllObjects cVector mtcars nMatrix
ObjectSize
```

This function returns a nested result. The first element is the name of the file: the default location is used as the path and the file name is unique, not least because it includes the timestamp: see lines [2] and [8].The second element returns the names of all the objects written to the file. If the active session is empty, both elements of the result are empty. The file can be loaded in the active session as follows:

```
      ⎕wi 'XEvaluateNoReturn' 'rm(list=objects());' ⍝ Optional - Clear active workspace
      ⎕wi 'XEvaluateNoReturn' 'source("C:/AJAY/RSTATS/SCRIPTS/apl2C_ALL20120320082113.R
");' ⍝ Load all definitions
```

Incidentally, a convenient way to delete a file in R, without sending it to the recycle bin, is this:

```
> unlink("c:/myMat.R"); # Rather counter-intuitive?
```

This command is silent: if the file does not exist, it does not raise an error.

6. Why use R?

Although R has APL like features, the two languages are not substitutes; therefore, any contention regarding the exclusive use of one or the other is irrelevant. R seems to be the ultimate self-contained personal computing tool: subject matter experts—for whom expertise in Statistics and Data Visualisation are inherent—simply use R as a tool of thought without regard for any technology paradigms or platform considerations.

For APL+Win users, R offers cost effective ways to incorporate R-like features in APL end-user systems. In other words, R does not produce applications for end users: the users are also the end-users, and they rely on either the command line interface or no interface at all (with R running scripts in batch mode). With R's COM server capabilities, some of the benefits for APL+Win users include the following:

- R provides access to a documented and debugged array of statistical analysis tools.
- The availability of powerful graphing capability and the incorporation of R produced graphs into any APL+Win driven graphical user interface.
- A formalisation of data structures in APL+Win, along the lines of structures that exist in R. At present, APL+Win data, especially heterogeneous nested data, is completely free form; that is, structures vary from application to application or even developer to developer.
- APL+Win can also provide an asynchronous interface to R via script and data files: these files are simple text files that follow simple layout patterns.
- APL+Win can utilize/acquire R data files and manage that data; this includes archiving, maintenance, and further processing in contexts where APL+Win may have superior capabilities.

In summary, consider the plausibility and feasibility of the following objectives:

- Enhance APL+Win applications by the incorporation of R as a collaborative and complementary tool.
- Simplify the R experience for users by seamlessly enabling the creation of R data structures and script files, thereby making it possible to use R in batch or non-interactive mode. APL+Win and R need not co-exist on the same computer.

6.1. No room for R?

Depending on the subject matter to which you apply APL, this might be a perfectly legitimate response: there is no room for R or there is no need to learn yet another technology. My personal view is that whether or not you find any use for R, you can only come to a conclusion after an in-depth look at this language.

- Whatever its design flaws, R is robust, free, and very well documented, albeit you need to learn to get accustomed to the available material. Initially, concentrate on sources which have worked examples that you can copy or reproduce and experiment with. R appears to have a more far-reaching momentum than APL.
- R is very close to APL in that it naturally copes with non scalar calculations. It provides a lot of features that correspond directly with the hallmarks of APL. In this respect, R provides a ready-made test environment for debugging or enhancing your APL code and coding style.
- The one compelling reason for using R is its capability with data visualisation, that is, graphs. In this respect, it is a deserving complement to APL which is lacking in such facilities. R's graphing capability is many times superior to those of Excel.

7. R on the internet

The single letter name of this language makes it cumbersome to locate material on the web. This is especially the case for new starters as they would not have acquired fluency with the R vocabulary, which is necessary for effective searching. However, there is a wealth of material available. Try the following from an R session to restrict search results:

```
> RSiteSearch("string replace",restrict='functions');
```

The first argument is the topic and the options for the second arguments are **functions|vignettes|views**.

The following site http://addictedtor.free.fr/graphiques/allgraph.php provides a lot of sample code—which you can copy and paste in your own R session—for exploring visualisation of data.

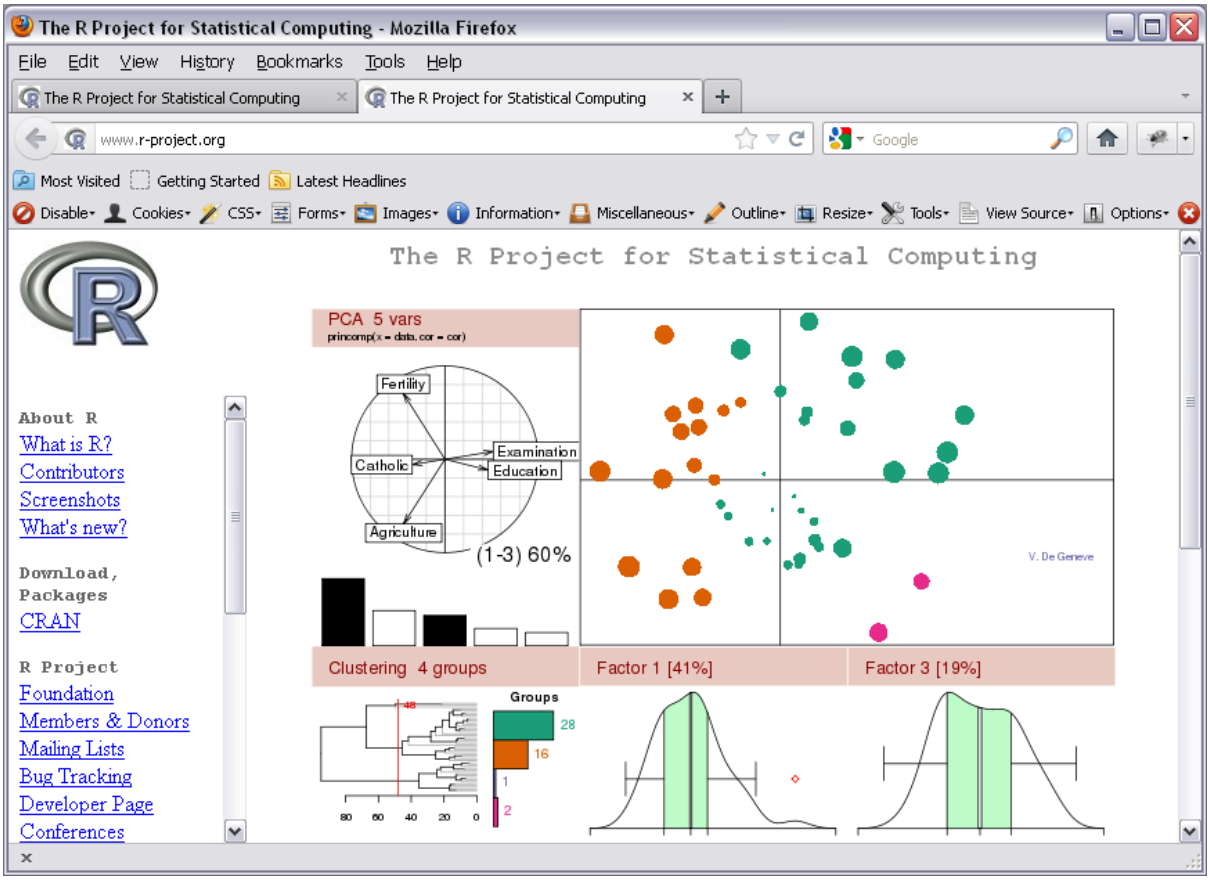
For a quick introduction or overview of the language, refer to http://www.statmethods.net/index.html. This reference is particularly suitable for programmers of other languages such as APL.

If inclined to do so, subscribe to the R mailing list—read forum—here: https://stat.ethz.ch/mailman/listinfo/r-help. As an alternative, consider subscribing to http://stackoverflow.com/questions/tagged/r. Forums are a particularly good and focussed source of inspiration as they provide worked solutions to actual problems encountered by users. Also, they provide access to free help on demand; you can post your own questions or problems and generally responses are forthcoming within minutes. A word of warning: be sure to follow the forum's posting guide since the regular contributors tend to be rather unforgiving, that is, dismissive of trespasses.

Alternatively, simply use the following link to browse the R mailing list - http://groups.google.com/group/r-help-archive/topics.

7.1. CRAN: Comprehensive R Archive Network

All versions of R and its documentation are available at http://www.r-project.org/. This is the obvious starting point for fact-finding about R. The site has all the links to manuals and provides links to every aspect of R. When browsing the site, click on the picture to see the source code for all the plots displayed.



7.2. Who uses R?

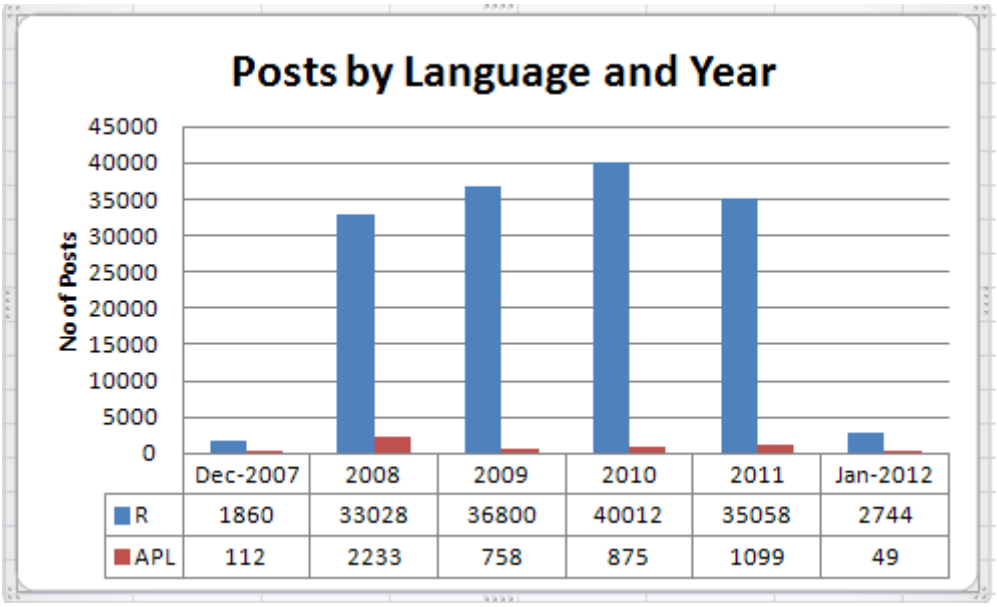
The similarity between APL and R—in terms of core language functionality—is simply astounding.

Some way into this project, I wondered about the user base and support that R has. This language became available towards the end of 1997.

An insight into the user base of R is truly eye-opening!

- http://en.wikipedia.org/wiki/R_(programming_language) provides a brief overview of the language; even if you have but a casual interest in this area, read this article.
- At http://www.r-project.org/, navigate to the **Members and Donors** link: this is indicative of the pedigree of the sponsorship and funding that R attracts.
- At http://rwiki.sciviews.org/doku.php?id=rugs:r_user_groups you will find a list of user groups by country.
- At http://www.dataspora.com/2009/02/predictive-analytics-using-r/ you will find details of how two well-known organisations are using R. This site also addresses some basic questions regarding the viability of R.
- At http://www.warwick.ac.uk/statsdept/user-2011/participant-list.html, you will find a partial list of attendees at the 2011 conference; the list contains 430 names. There were 465 attendees at the 2010 conference.
- At http://groups.google.com/group/r-help-archive/topics, you have read-only access to one of the R forums: if you pay some attention to the credentials of the correspondents, it becomes quite clear that R has a firm stronghold not only in commercial but also academic circles.

A simple indicator of R's success is a comparison of the number of posts recorded for APL at http://groups.google.com/group/comp.lang.apl/topics and those for R at http://groups.google.com/group/r-help-archive/topics: see the graph below.



8. Installation

This section reproduces the pages found at http://rcom.univie.ac.at/download.html. I expressly acknowledge **all** the intellectual property and copyright of the publishers/owners; I have reproduced the material purely for illustration and in order to highlight a simple route to getting started with R and APL+Win OLE/COM investigation.

You will need a working internet connection during the installation of the required software. For a minimal installation on the Windows platform[19], you need to power up your computer with administrator's privileges. At this stage, in order to keep things simple, all downloads from this site contain Windows binary–rather than source—files.

- First, download and install RAndFriendsSetup2142V3.2-4-1; I recommend that you select every option when given the choice.
- Second, review the licensing terms of all the products and ensure that you are able to abide fully by the terms and conditions; if not, you will need to uninstall the software.

The whole installation can be removed via **Start | Control Panel | Add or Remove Program**. Install all the components under a single directory e.g. c:\Program Files\R\ as this is handy if you have to remove the installation manually: you simply delete that directory[20].

The material at http://rcom.univie.ac.at/download.html, at the time of writing, follows; refer to the actual site as its content will no doubt be subject to ongoing changes to keep it up to date. It is possible that a later version of R is available—R version 2.15 is expected to appear at the end of March 2012. The portion following between the multicolour horizontal lines is reproduced from the site.

8.1. RAndFriends

This package will automatically install and configure:

- R 2.14.2
- rscproxy 1.3-1
- rcom 2.2-1

The R version contained in RAndFriends contains the 32bit and the 64bit version of R. On 32bit versions of Windows, only the 32bit version of R will be installed, on 64bit versions of Windows, both the 32bit and the

[19] Be guided by the web site as more up to date versions may be available.
[20] If you do this, you will need other tools to tidy up the registry and remove shortcuts and menu items.

64bit versions of R will be installed. All the statconn tools (statconnDCOM, rcom, RExcel ..) work *only* with the 32bit version of R, and RExcel works only with 32bit versions of Excel. RExcel is actively tested and supported for Excel 2003, Excel 2007, and Excel 2010.

RAndFriendsSetup will also download and install a suitable version of the statconnDCOM server and of RExcel during installation. Therefore you will need a working Internet connection during the installation process.

The installer named RAndFriends will (among other things) download the most recent release of the free noncommercial version of RExcel and the most recent release of the noncommercial version of statconnDCOM. The number after the V in RAndFriendsInstaller indicates the version of RExcel available at the release of RAndFriends.

If a new version of RExcel was released after the release of RAndFriends, the downloaded version of RExcel will have a version number higher than this number.

This version of RAndFriends was created 20120308.

Download RAndFriendsSetup2142V3.2-4-1

If you installed RAndFriendsSetup2120V3.1-9-1, you might have problems uninstalling R 2.12.0 contained in this setup. If that is the case, follow these instructions to uninstall this version of R.

We also give you information how to download all sources for R and the R packages included in RAndFriends.

8.2. RExcel

The current version is RExcel 3.2.4. It works only with 32bit versions of Excel (which can be installed on 64bit versions of Windows). The Excel versions supported are 2003, 2007, and 2010.

RExcel 3.2.4 requires R 2.12.0 (or later). The foreground server might work with earlier versions of R, but we do not officially support this.

The easiest way to install RExcel is to use the package 'RExcelInstaller' from CRAN. To install RExcel this way, you have to start R as administrator. On Windows XP you should be logged in as administrator, on Windows Vista and Windows 7 right-click on the R icon and select 'Run as administrator'.

In R, run the following commands:

- install.packages("RExcelInstaller")
- library(RExcelInstaller)

You will get further instructions on how to install in RExcel. You can also install RExcel without the R package (RExcelInstaller), although this is not the recommended way.

You will have to install

- a suitable version of R
- a matching version of rscproxy
- statconnDCOM or rcom with statconnDCOM

Download RExcel 3.2.4

Download REXCEL NONCOMMERCIAL USE LICENSE

8.3. SWord

This package contains SWord 0.99-3B3Beta.

You will have to install:

- a suitable version of R
- a matching version of rscproxy
- statconnDCOM or rcom with statconnDCOM

Download SWord 0.99-3B3Beta

Download SWORD PUBLIC LICENSE

8.4. statconn.NET

statconn.NET is not yet available. We provide an early version of the documentation for download as a ZIPped CHM file.

8.5. statconnDCOM

This package contains statconnDCOM3.3-0B2.

You will have to install:

- a suitable version of R (>= 2.12.0)
- a matching version of rscproxy

Download statconnDCOM 3.3-0B2_Noncommercial.

Download STATCONN NONCOMMERCIAL USE LICENSE .

8.6. statconnWS

This package contains statconnWS0.8-0B5Beta for Windows.

You will have to install

- a suitable version of R (>= R 2.12.0)
- a matching version of rscproxy

This beta/test version expires 11-10-01. It will stop working then

Download statconnWS 0.8-0B5Beta.

8.7. ROOo

ROOo is an OpenOffice.org extension. The current version (0.751) works on Mac OSX (10.5 or higher), Windows (XP, Vista or Windows 7), and Linux x86 (32bit as well as 64bit). To run ROOo you need:

- a suitable version of R (>= 2.12)
- a matching version of rscproxy
- OpenOffice.org 3.0 or higher (ROOo has been testes with LibreOffice and NeoOffice too)

Uninstall any previous version of ROOo by removing the componentes ROOo.oxt, StatConnector.oxt, ROOoMacro.oxt, RInterface.oxt from the OpenOffice.org extension manager (Tools -> Extension manager, select the component and press Remove). Restart OpenOffice.org after removing the old extensions. To install the current version of ROOo, simply open the extension manager of OpenOffice.org (Tools -> Extension manager), press Add and select the file ROOo.oxt. When the installation has been completed sucessfully, verify the ROOo options (Options -> ROOo -> Path Settings). Depending on the operating system, several properties are available:

4. Mac OSX: Verify the directories configured for R-Home and Proxy. Modify these settings if required. You may also choose between the 32bit and the 64bit version of R.
5. Windows: No options available. R-Home is determined by reading the Windows registry. Currently only the 32bit version of R is supported by ROOo.
6. Linux: Verify the directories configured for R-Home and Proxy. Modify these settings if required. 64bit Linux requires 64bit versions of OpenOffice.org and R, hence the 64bit option cannot be changed for the Linux version of ROOo.

This package contains ROOo 20090112-1 Beta.

You will have to install

- a suitable version of R
- a matching version of rscproxy

Download ROOo.oxt.

8.8. Scilab within Excel

An Excel workbook demonstrating how to use Scilab directly from within Excel. This requires statconnDCOM. The statconnDCOM documentation explains how to install the components needed to access Scilab through statconnDCOM.

Download ScilabTest.xls

8.9. Creating and Deploying an Application with (R)Excel and R

This is the example code for the article *Creating and Deploying an Application with (R)Excel and R: Recursive Partitioning and Regression Trees*. Requires statconnDCOM and RExcel. For a new installation, you should consider downloading RAndFriends.

- Download RExcelRpartDemo.zip (demo worksheets)
- Download RExcelrpart_1.0.tar.gz (R source package)
- Download RExcelrpart_1.0.zip (R binary package for R 2.10.0, Windows)

8.10. Post installation verifications

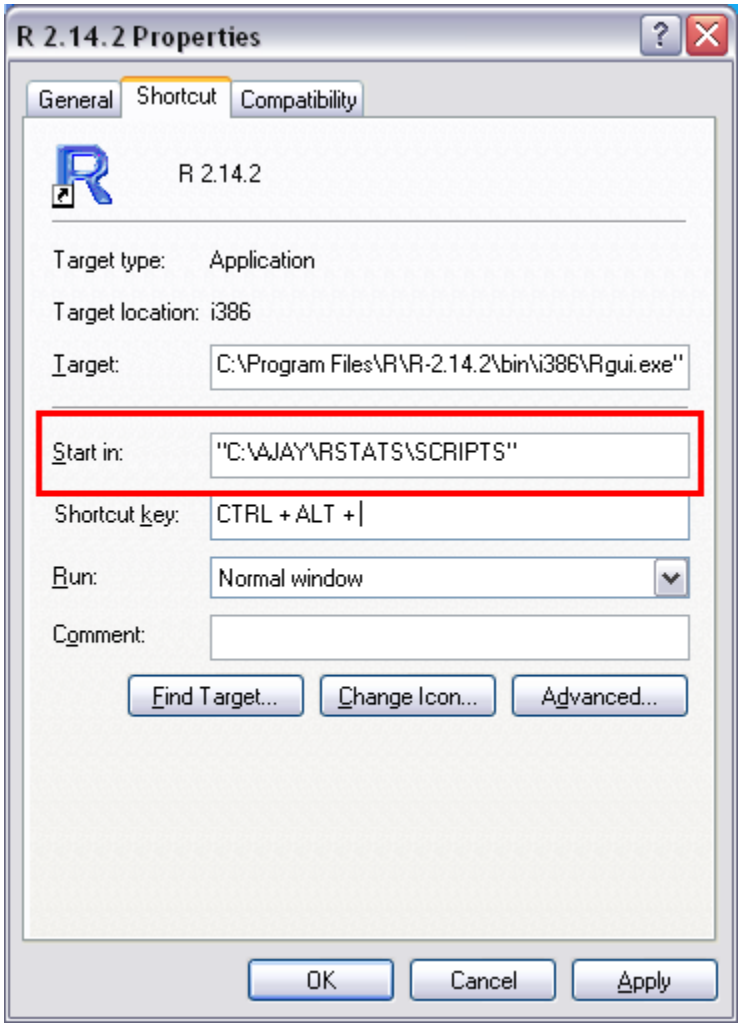
The installation procedure suggested above is hassle free; it installs R, RExcel, Sword, and several other components ensuring compatibility between R and the constituent components. This process also takes care of registry entries and configures R to start-up in ready mode for OLE/COM client/server tasks.

I am using APL+Win version 11.1 on a computer running Windows XP with Service Pack 3. Following installation, launch R to see it in interactive mode.

I have highlighted the portions of the interface that details the version, platform and confirmation that the packages **rcom** and **rscproxy** are loaded: the latter packages are essential for the coupling of R and APL+Win in OLE/COM client/server roles.

8.10.1. Customizing the R shortcut

The next step is to choose the location for storing all R related files. The general recommendation is that you modify the start in location in the R shortcut.



The recommended method is to modify the R shortcut, in particular, the 'Start in' option. The path that you specify must already exist.

Incidentally, you can execute the following command from an R session to create a path with ease:

```
> dir.create("c:/a/b/c/d/e/")
```

If the argument comprises a valid path, that is, it does not contain any illegal characters R will create the whole tree irrespective of whether none or any node in the path exists.

This is equivalent to the Win32 API call MakeSureDirectoryPathExists.

Win32 API DLLs are inaccessible from R unless you write your own wrapper around the API calls or simply use APL+Win as server and call the API from the server.

Ideally, you should store files relating to each project in a distinct folder of your choice.

8.10.2. R as the OLE/COM client for an APL+Win server

R can start a COM server in two ways; the next step is to start APL+Win as the COM server and ensure that this is working.

The first way:

```
> aplWin<-comCreateObject("APLW.WSEngine")
> aplWin<-comCreateObject("APLW.WSEngine");
> aplWin[["Visible"]]<-TRUE;
> comInvoke(aplWin,"Exec","2+3 4 5");
[1] 5 6 7
```

This confirms that the client and server can indeed communicate.

The second way:

```
> library(SWinTypeLibs);
Loading required package: RDCOMClient
> apl<-COMCreate("APLW.WSEngine");
> apl[["Visible"]]<-TRUE;
> .COM(apl,"Exec","2+3");
[1] 5
```

Note that a package SWinTypeLibs needs to be loaded before invoking APL+Win as the Server. The installation of this package ensures that all the packages it depends on are loaded automatically.

The general syntax of the two methods of invoking a COM server is very similar; the keywords are different and the second method as the overhead of additional packages being required. The important point is that we have successfully used APL+Win as a server to an R client in two different ways.

By the way the following displays all the libraries available.

```
> library();# no argument specified
```

The library with my installation is located at C:/Program Files/R/R-2.14.2/library.

A library is unloaded—removed from the search path—as follows.

```
> detach(package:SWinTypeLibs);
```

8.10.3. R as the OLE/COM server for an APL+Win client

The following confirms that APL+Win can use R as a COM Server.

```
    ∇ R
[1]   ⍝ Ajay Askoolum - APL2000 Conference April 22-24, 2012
[2]   '#' ⎕wi 'ReleaseObjects'
[3]   '#' ⎕wi 'Reset'
[4]   ⎕wself←'RServer' ⎕wi 'Create' 'StatConnectorSrv.StatConnector'
[5]   ⎕wi 'XInit' 'R'
    ∇
```

The enumeration of the properties, methods, and events of the server provide confirmation and the starting point for investigation.

```
      ('x'=↑¨r)/r←⎕wi 'properties' ⍝ There are none
      6 3⍴('X'=↑¨r)/r←⎕wi 'methods' ⍝ There are 18
 XAddGraphicsDevice    XClose                      XEvaluate
 XEvaluateNoReturn     XGetConnectorInformation    XGetErrorId
 XGetErrorText         XGetInterpreterInformation  XGetServerInformation
 XGetSupportedTypes    XGetSymbol                  XInit
 XRemoveGraphicsDevice XSetCharacterOutputDevice   XSetErrorDevice
 XSetSymbol            XSetTracingDevice           XSetUserInterfaceAgent
      ('X'=↑¨r)/r←⎕wi 'events' ⍝ There are none
```

8.10.3.1. R Server characteristics

The instance of the server does not expose a 'Visible' property; therefore, the Server cannot be made visible easily, as instances of Excel.Application and APLW.WSEngine can be. Core R does not have a user interface; therefore it cannot be made visible.

The instance does not expose any properties or events. The methods that are exposed are listed below. The documentation of the COM interface is severely lacking in enumerating the possible arguments of the methods. The documentation is found at http://cran.r-project.org/web/packages/rcom/rcom.pdf and http://cran.r-project.org/web/packages/rscproxy/rscproxy.pdf

The methods I aim to investigate are shown with a shaded background; they provide the functionality for investigating the potential for using R.

```
XAddGraphicsDevice

The syntax is:

pDevice@Object_ISGFX ← ⎕WI 'XAddGraphicsDevice' bstrName@String pDevice@Object_ISGFX
```

```
XClose

The syntax is :

⎕WI 'XClose'

Terminates the R Session.
```

```
XEvaluate

Evaluates a valid R expression and returns the result to APL+Win.
```

```
XEvaluateNoReturn

Evaluates a valid R expression and does not return any result to APL+Win.
```

```
XGetConnectorInformation

The syntax is:

Result@String ← ⎕WI 'XGetConnectorInformation' lInformationType@Long_InformationType

XGetErrorId

The syntax is:

Result@Long ← ⎕WI 'XGetErrorId'

XGetErrorText

The syntax is:

Result@String ← ⎕WI 'XGetErrorText'

XGetInterpreterInformation

The syntax is:

Result@String ← ⎕WI 'XGetInterpreterInformation' lInformationType@Long_InformationType

XGetServerInformation

The syntax is:

Result@String ← ⎕WI 'XGetServerInformation' lInformationType@Long_InformationType

XGetSupportedTypes

The syntax is:

pulTypeMask@Long ← ⎕WI 'XGetSupportedTypes' pulTypeMask@Long

Returns information about types supported by the server.

XGetSymbol

The syntax is:

Result ← ⎕WI 'XGetSymbol' bstrSymbolName@String

Retrieves the value of a symbol from the server or generates an error if the symbol does
not exist.

XInit

The syntax is:

⎕WI 'XInit' bstrConnectorName@String

 ⎕wi 'XInit§ R ⍝ Initializes the R Language

XRemoveGraphicsDevice

The syntax is:

⎕WI 'XRemoveGraphicsDevice' bstrName@String

XSetCharacterOutputDevice

The syntax is:

pCharDevice@Object_IStatConnectorCharacterDevice ← ⎕WI 'XSetCharacterOutputDevice'
pCharDevice@Object_IStatConnectorCharacterDevice
```

```
XSetErrorDevice

The syntax is:

pCharDevice@Object_IStatConnectorCharacterDevice ← ⎕WI 'XSetErrorDevice'
pCharDevice@Object_IStatConnectorCharacterDevice

XSetSymbol

Creates or resets a symbol in the Server. The syntax is:

 ⎕WI 'XSetSymbol' bstrSymbolName@String vData

This method requires two arguments, a valid name and its value specified as a Variant
type.

XSetTracingDevice

The syntax is:

pCharDevice@Object_IStatConnectorCharacterDevice ← ⎕WI 'XSetTracingDevice'
pCharDevice@Object_IStatConnectorCharacterDevice

XSetUserInterfaceAgent

The syntax is:

pUIAgent@Object_IStatConnectorUIAgent ← ⎕WI 'XSetUserInterfaceAgent'
pUIAgent@Object_IStatConnectorUIAgent
```

When the function that creates the instance of R as a server is run, the following pop-up appears momentarily; it provides further details about licensing terms and options for licensing. During the course of electronic mail exchanges, the copyright holder was at pain to point out that the basic installation permits non-commercial use only.

Copyright (C) 1998-2012 by Thomas Baier.

This version of statconnDCOM may be used under the terms of the
STATCONN DCOM NONCOMMERCIAL USE LICENSE.
See the file SC_PUBLIC in the installation folder for the full license terms.

All publications based on analyses performed using statconnDCOM directly or indirectly (e.g. RExcel, FlexArray, SAM,...) will include the following citation:

Baier Thomas & Neuwirth Erich (2007). Excel :: COM :: R.
Computational Statistics, Volume 22, Number 1/April 2007. Physica Verlag.

Please visit **http://www.statconn.com/** for alternative licenses.

Support for this version of statconnDCOM is available only through http://rcom.univie.ac.at/ (Wiki, Mailing List)

8.11. R tour

The basic installation of R incorporates a handy method for rapidly gaining a sense of R. The following command invokes a graphical interface that lists a series of pre-prepared demonstrations:

```
> demo();
```

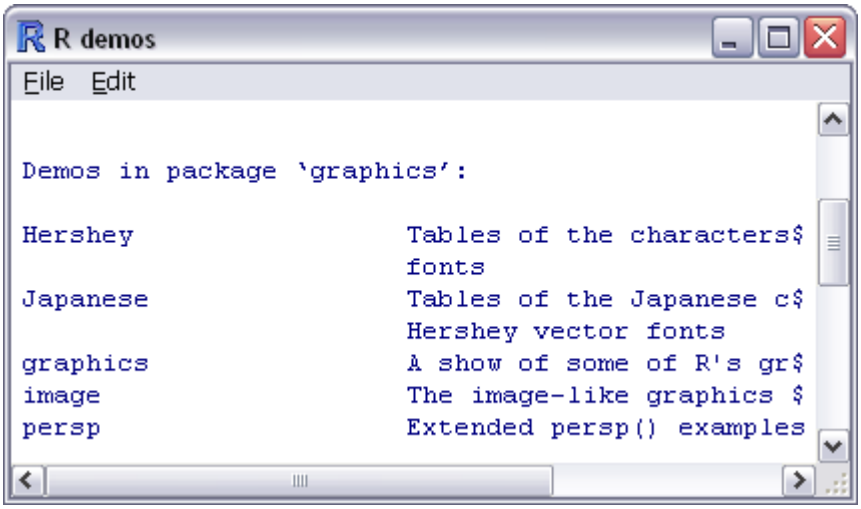
Review each demonstration using the following syntax:

```
>demo(graphics);
```

The argument is the name of the package; there are several packages listed, see below. The demonstrations are robust and provide working code that you can copy and paste into the R session; this promotes confidence in tackling the R language.

For a serious review of R you need some language manuals, if only to gain an comprehensive overview of R. Refer to the Installation section for details of where the manuals for the language are available.

- The demonstrations are fully interactive.
- All the code that produces what is shown is listed, together with comments, in your R session.



Use history(Inf) to capture the session and then you can run the code with or without customised modifications.

9. Conclusion

I use C# and SQL Server in the domains of pensions and maritime navigation to put the daily bread on the table. I have no particular expertise in statistical analyses or data visualisations. Now that I can also put R on my Curriculum Vitae, I'll share my own thoughts about the language.

- R, like APL, has avid and dedicated advocates and this creates its own comfort zone. Therefore, it tends to be used for purposes such as data cleaning—because its practitioners know R—prior to doing the actual analysis. APL is a more versatile general purpose language, not least because APL is capable of producing fully-fledged applications with graphical user interfaces.
- R and APL+Win introduce complexities in application design because, by default, their programming paradigm does not promote tiered application design, especially the separation of the data tier.
- R needs something equivalent to 'C# LINQ to Objects', that is, a more generic approach to in-memory data cleaning and analysis package.
- R is a valuable skill to add to my skills set; this proved to be fairly effortless since R is in many respects like APL. In other words, with knowledge of APL, no leap of faith was necessary to see R extend naturally to cope with non-scalar data. The biggest hurdle in learning R is in getting accustomed to its jargon, for example, sub setting means selection and compression in APL.
- APL is a valuable skill to bring to R; coding R in APL style is a fairly natural way to ease into the R language. For a concrete example, refer to http://escholarship.org/uc/item/3kw1196f.
- I will use R for incorporating data visualisation into applications; I do not know of any other package that is free and capable of producing publication quality graphs with so little effort.
- Most of R's analytical capabilities are based on powerful primitive functions and a handful of specific and portable data structures, unlike APL where data structures—nested vectors and arrays—are non-portable and free form. For APL applications, I will investigate the collation of data into structures that optimize performance and opportunities especially when combined with utility functions tailored for purpose. Consider an example where diligently or appropriately coded utility functions reduce the coding overhead:

```
Example<-function(){
        # Simulates rolling two dice
```

```
        numrolls <- 1000000;
        die1 <- sample(6,numrolls,TRUE); # Like APL 6?1000000 but with replacement
        die2 <- sample(6,numrolls,TRUE);
        print(mean(die1 > die2)); # estimated probability die1 greater than die2
        print(mean(die1 + die2)); # estimated expected value of sum of rolls
        print(table(die1 + die2)/numrolls); # estimated distribution of total of two rolls
}
> timeTaken<-system.time(Example());
[1] 0.416113
[1] 6.997967

       2        3        4        5        6        7        8        9
0.027662 0.055934 0.083610 0.111043 0.138424 0.166962 0.138433 0.111544
      10       11       12
0.083253 0.055602 0.027533
> timeTaken; #Seconds
   user  system elapsed
   1.34    0.04    1.39
```

Of course, you can avoid the coding overhead altogether if the solution is accessible from your environment.

```
      ⎕wi 'XSetSymbol' 'myNums' (10?1000)
      ⎕wi 'XGetSymbol' 'myNums'
356 741 973 13 790 1000 438 619 873 546
      ⍕⎕wi 'XEvaluate' 'capture.output(summary(myNums))'
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
   13.0   465.0   680.0   634.9   852.2  1000.0
```

I recommend that you take a closer look at R and especially as a server application to an APL+Win client; your experience with R will yield worthwhile and lasting dividends in your APL+Win application design and development.

9.1. Lest I forget

- Thank you Varuna for proof-reading and suggesting changes so that what I had written make more sense; Varuna is my daughter.
- Thank you Joe Blaze for the technical review and feedback.
- The codes—APL+Win and R—I have used in this paper do not include error handling; for this reason alone, the code is not meant for use in production systems.
- If you have any comments, please post your thoughts on the http://www.apl2000.com forum.

Ajay Askoolum

March 2012

References

Repositories & Manuals

http://cran.r-project.org/	The Comprehensive R Archive Network
http://www.r-project.org/	R Manuals
http://cran.us.r-project.org/doc/manuals/R-intro.pdf	R Introduction
http://cran.us.r-project.org/doc/manuals/R-data.pdf	Data Import/Export
http://cran.us.r-project.org/doc/manuals/fullrefman.pdf	Full Reference R Manual
http://rwiki.sciviews.org/doku.php?id=rugs:r_user_groups	User groups by country
http://rcom.univie.ac.at/	General Information
http://www.omegahat.org/	Other download details
http://www.londonr.org/	UK's User Group[21]
http://faculty.washington.edu/tlumley/Rcourse/R-fundamentals.pdf	Tutorial on using R
http://addictedtor.free.fr/graphiques/allgraph.php	R Graph Gallery with Source Code

Mailing Lists[22]

http://groups.google.com/group/r-help-archive/topics	R Mailing List

Downloads[23]

http://rcom.univie.ac.at/download.html	Simplified way to install R
http://www.omegahat.org/R/bin/windows/contrib/2.14/[24]	Other packages for R 2.14.2
http://cran.r-project.org/contrib/extra/dcom/	DCOM for R 2.14.2

userR! 2012 Conference (12–15 June)

http://www.r-bloggers.com/call-for-abstracts-for-user-2012/

R for Windows: Frequently Asked Questions

http://cran.r-project.org/bin/windows/rw-FAQ.html	See 2.18 - Does R support automation (OLE, COM)?[25]

R Journals

http://journal.r-project.org/	Includes archive of back issues

Notes:

- At the time of writing, I am using the latest available version, namely, R for Windows version 2.14.2. I downloaded the binary files in order to keep matters simple; alternatively, you have the option to download the source and re-compile.
- If installing on Windows 7, you need to install and run R with administrator privileges for exploring R without shackles.
- This above is a basic list of reference materials on the installation and usage of R. The internet has many more references on specific aspects of R.
- You will need to review the licensing terms for R and each package individually before any commercial deployment of applications based on them.

[21] This user group has its own mailing list and offers commercial courses.
[22] This is like forums–you need to register prior to being able to post questions or replies.
[23] There are dedicated mailing lists for issues relating to packages from independent package contributors.
[24] FTP site is organised by version. Get versions of package(s) from the most recent version when unavailable in version 2.14.
[25] "Directly, no. See CRAN packages rscproxy and rcom as well as RDCOMServer, RDCOMClient, RDCOMEvents and SWinTypeLibs from http://www.omegahat.org/."

Index