Statistical computing based on Dyalog APL and the R statistical system

Alexander Skomorokhov and Vladimir Kutinsky
Institute of Nuclear Power Engineering
P.O. Box 5061, Obninsk-5
Kaluga Region, 249020, Russia
askom@obninsk.com, kutinskyv@obninsk.com

Abstract
The present paper describes the implementation of a simple interface between Dyalog APL and R programming language in order to build a powerful tool for statistical computations and complex data analysis. Among the topics discussed in the paper are the following: the main steps to build the interface by means of a DCOM server, examples of using the interface including calls to simple R function from within the Dyalog APL session, graphical representation of different kinds of plots including 3D plots, the examples of the joint use of R libraries and Dyalog APL. As the conclusion, an example of a more complex tree based data analysis and the comparative analysis of the use of similar functions written in Dyalog APL are discussed.

Introduction
The APL programming language is the array programming language. In other words, it makes it much easier to work with large data sets by means of a number of built-in operators, dynamic functions and other useful facilities that simplifies the code and the work itself. In fact, this is the very data format every statistician deals with in his daily tasks. This is why APL is so attractive as a possible platform for statistical computing.

Unfortunately, nothing is perfect, as everything in this world. Every software product is usually concentrated on some specific tasks and those that claim to offer generality are often too complex, hulky and sometimes quite expensive. The obvious solution to this problem is in combing the best parts of different software products and building a joint powerful tool that will suit one’s specific tasks.

We and many of our colleagues are devoted users of APL. On the other hand, in our work we often have to deal with different sorts of tasks that imply the use of statistical algorithms and models. At present, the best statistics oriented software product, in our opinion, is the R language [1], [2]. R is `GNU S’ - a language and environment for statistical computing and graphics. This system is freely available for major platforms, including Windows and Linux operating systems. R is similar to the award-winning S system, which was developed at Bell Laboratories by John Chambers et al. It provides a wide variety of statistical techniques (linear and nonlinear modeling, statistical tests, time series analysis, classification, clustering, etc.) and graphical techniques (histograms, box-whiskers plots, 3D plots etc).

Below are at least three reasons that made us try to link the R statistical system and APL:

Functions library. R comes with a large number of ready-to-use functions to make different kinds of statistical calculations that are available either in its base distribution or in separate packages. So, one doesn’t have to slave at books looking for the algorithms and then implement them. All one needs is to call the appropriate R function.

Graphics. A data analysis cannot be done properly unless one uses graphics that makes it easier to interpret the results, make further decisions and share the results with others. R graphics speaks for itself.
Speed of computations. Calls to R functions are in fact calls to R dynamic libraries (DLLs) that work much faster than the similar APL codes.

Thus, APL with its power to handle data and the R statistical system are those two parts we will use to create a powerful tool for statistical computing.

Building the Interface

As it was said above, the interface between Dyalog APL and R is done via the DCOM server. DCOM is a Windows protocol for communicating between different programs, possibly on different machines. Thomas Baier's StatConnector program available from http://cran.r-project.org/contrib/extra/dcom provides an interface to the proxy DLL which ships with the Windows version of R and makes a DCOM server. This can be used to pass simple objects (vectors and matrices) to and from R and to submit commands to R.

Assuming that you have successfully installed R and DCOM and you can see a registered library for the DCOM server (StatConnectorSrv 1.0 Type Library) in the Dyalog APL Workspace Explorer, you can create an OLEClient namespace:

```
'#.R'ŒWC'OleClient'EE
'StatConnectorSrv.StatConnector'
```

Now, we can explore the main methods exposed by the OLE server:

- `R.Init('R')` – to fire up R;
- `R.Close` – to shut down R;
- `R.SetSymbol('symname',value)` – to pass APL data value to R where it will have the name symname;
- `y РФ R.GetSymbol('symname')` – to pass data from R to APL;
- `y РФ R.Evaluate('expression')` – to submit commands to R. The result will be saved in y;
- `R.EvaluateNoReturn('expression')` – to submit commands to R that do not return explicit results (assignment, plotting graphics etc).

As we will show later, using only these methods can give us the desired access to R functions.

Using the Interface

Let us start with a simple example. Suppose we have a vector of values and we would like to know the density for the normal distribution for these values. In R there is a function `dnorm(x, mean=0, sd=1,..)` that does the job. Here, `x` is a vector of values, `mean` is the value of the mean (default is 0); `sd` is the value of the standard deviation (default is 1); .. indicates other arguments that are not relevant now. Let us use this function. First of all, we have to create an OLEClient namespace and fire up R:

```
'#.R'ŒW'OleClient'EE
'StatConnectorSrv.StatConnector'
R.Init('R')
```

Now, we need to pass `x` to R. We will use the same name for that variable:

```
+x РФ 10?10
9 5 8 10 1 4 2 6 7 3
R.SetSymbol('x',‹x)
```

We can explore what variables are available for R by means of the R command `ls()` returning the names of all variables (in fact objects) and view their values:

```
R.Evaluate('ls()')
x
R.Evaluate('x')
9 5 8 10 1 4 2 6 7 3
```

As it is shown above, the R session now contains our variable `x` that is identical to our APL variable `x`. The last step is to call the R function `dnorm()` and return the result to the Dyalog APL session:

```
R.Evaluate('ls()')
x
R.Evaluate('x')
9 5 8 10 1 4 2 6 7 3
R.EvaluateNoReturn('d РФ dnorm(x)')
R.Evaluate('ls()')
d x
d РФ R.GetSymbol('d')
10
```

The APL variable `d` now contains the values of the density of the normal distribution for the corresponding values of `x`.

While this is a very simple and unexciting example, it is powerful. You can see how easily we can perform data interchange between the APL and R sessions, call R functions, explore the results and return the results back to the APL session.
To simplify the work with the R-APL interface we wrote a few cover functions. The function \texttt{rinit} initializes the R-APL interface:

\begin{verbatim}
[0] rinit
[1] '#.R'\[WC'OleClient'
    'StatConnectorSrv.StatConnector'
[2] #.R.Init(’R')
\end{verbatim}

The function takes no arguments, creates an OLEClient namespace \#.R (line [1]) and fires up R (line [2]).

The function \texttt{rput} is used to pass variables from the Dyalog APL session to R:

\begin{verbatim}
[0] {m}rput
[1] -(0=\[NC’m’)/’m”’x’'
[2] m+,cm
[3] a+cm
[4] #.R.SetSymbol(m,a)
\end{verbatim}

The right argument is the data to pass to R and the optional left argument (a character vector) is a name the data will have in R. By default, the name of the variable is \texttt{x} (line [1]).

The function \texttt{rget} is used to pass data from R to the Dyalog APL session:

\begin{verbatim}
[0] r+rget name
[1] r+#.R.GetSymbol(name)
\end{verbatim}

The only argument it takes is a character vector indicating the name of the R variable we want to get.

The function \texttt{rexec} executes an R command:

\begin{verbatim}
[0] a+(r)rexec exp
[1] *(0=\[NC’r’)/’r”’0'
[2] :If r
[3] a+#.R.Evaluate(exp)
[5] #.R.EvaluateNoReturn(exp)
[6] :EndIf
\end{verbatim}

The right argument of the function is a character string (vector) containing the R command. The optional left argument indicates whether or not the function returns a result.

From now on we will use these cover functions to perform data exchange or to call R functions.

Let us now consider a more complex example. Suppose we have a numeric matrix and we would like to calculate the correlation between its columns. To fulfill this task we will get use of the R function \texttt{cor}(x,y) which calculates the correlation between its arguments. Below is an example matrix \texttt{x}:

\begin{verbatim}
x<-\{3
    3
    1
    1
    0
    50
    -2
    0
    -2
\}
x
\end{verbatim}

We will use cover functions \texttt{rput} to pass the matrix to R and \texttt{rexec} to call the function \texttt{dim()} to check the shape of the matrix:

\begin{verbatim}
’matr’ rput x
+b+1 rexec ’dim(matr)'
3 3
\end{verbatim}

As you can see the matrix \texttt{x} was passed to R and it preserved its shape. Let us now find the correlation between its columns:

\begin{verbatim}
1 rexec ’c1<-cor(matr[,1],matr[,1])’
1 rexec ’c2<-cor(matr[,1],matr[,2])’
0.755928946
1 rexec ’c3<-cor(matr[,1],matr[,3])’
0
\end{verbatim}

Finally, we can get those values back to APL:

\begin{verbatim}
+c1+rget ’c1'
1
+c2+rget ’c2'
0.755928946
+c3+rget ’c3'
0
\end{verbatim}

So, we successfully passed a simple numeric APL matrix to R, made the necessary calculations there and got the results back to APL.

Later, we will show how complex nested matrix can be passed to R.

**Graphics**

Let us get back to our first example when we calculated the density of the normal distribution for a vector of values. We deliberately didn’t print the values of \texttt{d} for it is more interesting to represent it graphically. This is what we are going to do now. First, to make the plot look a bit better, let’s change the values of \texttt{x} and make it a growing sequence of values:

\begin{verbatim}
x<-5.1+0.1*1:101
\end{verbatim}
Now, everything is ready to calculate the density of the normal distribution for $x$ and plot the result (Figure 1):

```r
'x' runif x
reex 'd<-dnorm(x)
reex 'plot(x,d,type="l",xlab="X", ylab="Density", main="Normal Distribution")'
```

![Normal Distribution](image1.png)

**Figure 1.** The plot of the density of the normal distribution for $x$.

As a conclusion to this section, we will demonstrate how 3D plots can be produced. The next example is taken from the help page for the function `persp()` that draws perspective plots of surfaces over the x–y plane (Figure 3):

```r
reex 'x<-seq(-10,10,length=50)
reex 'y<-x'
reex 'foo<-function(x,y)(r<-sqrt(x^2 +y^2)); 10*sin(r)/r'
reex 'z<-outer(x,y,foo)
reex 'persp(x,y,z,theta=30, phi=30, expand=0.5, xlab="X", ylab="Y", zlab="Z")'
```

![3D Plot of Surface](image3.png)

**Figure 3.** Perspective plot of a 3D surface.
Apart from the fact the plot looks impressive, there are a few things we would like to point out. First, as you can see we didn’t pass any variables to R but created them there directly (variables x and y). Finally, we can easily write functions in R and use them as if we work in the R session (the function \( \text{foo}(x, y) \)).

**Tree based models**

**R tree functions overview**

Now that we’ve shown the basics of the R-APL interface, it is time to consider a real complex example of data analysis, namely a classification tree based data analysis. Classification Trees are used to predict membership of cases or objects in the classes of a categorical dependent variable from their measurements on one or more predictor variables. The theory of classification trees can be found in [3]. The main advantages of tree based models, algorithms to grow classification trees, their APL implementation and applications are discussed in detail in [4]. In this paper we will concentrate on R functions for growing classification trees and their use from within the Dyalog APL session.

The functions for tree-based analysis are not part of the base R distribution but available as an add-on package “tree” that can be found in Package Sources at http://cran.r-project.org/. If the package “tree” is installed its functions are loaded into the R workspace by means of \texttt{library()} function that takes a name of the desired package to load as an argument. So, in order to load the “tree” functions we need to \texttt{library(tree)}.

The base function that grows a classification tree is \texttt{tree(data,formula,args)}, where \texttt{data} indicates the name of the data in the R workspace; \texttt{formula} defines a response variable and predictor variables the response variable depends on. The argument \texttt{arg} indicates other parameters to \texttt{tree()}.

As the real data are always “noisy”, the classification tree we get as a result of the \texttt{tree()} function is usually overfitted. The normal procedure is either to manually snip off unnecessary tree nodes (thus, to use one’s past time experience or intuition) or to apply a special procedure to find the right size for the tree. This procedure is known as the V-fold Cross Validation and is used to find the size of the tree that is neither underfitted nor overfitted. The R function for the Cross Validation procedure is \texttt{cv.tree(tree,FUN,method,..)}. It takes a number of arguments: \texttt{tree} is the classification tree to find the best size for; \texttt{FUN} is the function to do the pruning; \texttt{method} is a character string denoting the measure of node heterogeneity.

Applying the \texttt{cv.tree()} function to the initial tree we get a so-called cost complexity parameter \( k \) that is used in the function \texttt{prune.tree(tree, k, method)} to snip off the least important splits and thus get the tree of the right size.

**Input data**

For illustration purposes we will use the well-known Iris data set that was introduced by R.A.Fisher as an example for discriminant analysis. The data report four characteristics (sepal width, sepal length, petal width and petal length) of three species of the Iris flower (Setosa, Versicolor, Virginica). The data is represented in Table 1.

<table>
<thead>
<tr>
<th></th>
<th>( Y )</th>
<th>( V_1 )</th>
<th>( V_2 )</th>
<th>( V_3 )</th>
<th>( V_4 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setosa</td>
<td>49</td>
<td>30</td>
<td>14</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>Setosa</td>
<td>50</td>
<td>36</td>
<td>14</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>Versicolor</td>
<td>59</td>
<td>32</td>
<td>48</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>Versicolor</td>
<td>63</td>
<td>29</td>
<td>56</td>
<td>21</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
<tr>
<td>Virginica</td>
<td>77</td>
<td>30</td>
<td>61</td>
<td>23</td>
<td></td>
</tr>
<tr>
<td>Virginica</td>
<td>63</td>
<td>34</td>
<td>56</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td></td>
</tr>
</tbody>
</table>

Here, \( Y \) is a response variable and \( V_1 - V_4 \) are predictor variables.

In APL we represent continuous and ordered variables as simple numeric vectors. For categorical
predictors the natural representation is a nested character vector of value labels. The same is true for a response variable of class labels. The data set, shown in Table 1, is a nested matrix called *iris*:

\[
\begin{align*}
\text{Setosa} & \quad 51 \quad 35 \quad 14 \quad 2 \\
\text{Versicolor} & \quad 70 \quad 32 \quad 47 \quad 14 \\
\text{Virginica} & \quad 63 \quad 33 \quad 60 \quad 25
\end{align*}
\]

Growing the tree

The following cover function `rtree` is used to call R functions for growing classification trees for a given data set:

```r
rtree(data) = iris[1,51,101;]:
```

In line [14] we load the library `tree` into the R workspace so that all R functions to grow a classification tree and choose the right size for the tree are available. Then, in line [15] we use the cover function `rexec` to call the R function `tree()` to grow a classification tree. The last two arguments (minsize=2 and mindev=0) are used to grow a tree of the maximal size. The result `t` is an object of class 'tree'. This is a complex object and cannot be directly passed to the APL session. Fortunately, we don’t need the whole of it so we only take the first component of `t`, the matrix that depicts the structure of the result tree (line [16]). The names of the rows of this matrix are the numbers of the nodes of the tree. As far as we will need these values later we add an extra column containing them to the result (line [17]). To pass the result to APL we use the cover function `rget` (line [18]).

To simplify the code we used constant names for the data frame that holds the data set (`data`) and for the name of the tree we build (`t`).

Let us now use the function `rtree` to grow a classification tree for our data and explore the result:
The result \( t \) is a nested character matrix. Every row of it represents a node of the tree. The first number in each row of the output is a node number. The nodes are numbered to index the tree for quick identification. For a full binary tree, the nodes at depth \( d \) are integers \( n, 2^d \leq n < 2^{d+1} \). Usually, the tree is not full, but the numbers of the nodes that are present are the same as they would be in a full tree.

Let us examine one row of the output:

1. the node number
2. the name of the predictor that is used to split the node or ‘leaf’ if the node is terminal
3. the number of objects in the node
4. the value of deviance for the node (the measure of node heterogeneity)
5. the name of the class the objects in the node belong to
6. the threshold for the left node
7. the threshold for the right node

The last three columns should be regarded as a single column that represents the estimated probabilities of the classes in the corresponding nodes.

Let us now use the generic functions \texttt{plot()} and \texttt{text()} to plot the tree (Figure 4):

\begin{verbatim}
rexx 'plot(t)' 
rexx 'text(t)'
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{tree.png}
\caption{A Classification Tree.}
\end{figure}

The size of the tree is 9 and determined by the number of leaves or terminal nodes.

**Cross Validation**

Let us apply a 10-fold cross-validation procedure to find the right size for the tree and plot the result of the cross-validation (Figure 5):

\begin{verbatim}
rexx 'cv<-cv.tree(t,FUN=prune.tree, method="misclass")'
rexx 'plot(cv, type="b)'
\end{verbatim}

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{cv.png}
\caption{Cross Validation results.}
\end{figure}

As shown in this graph, the CV costs (the misclassification errors) approach their minimum quickly as tree size initially increases, but actually stop decreasing largely as tree size grows further. The right size of the tree is close to the point where the initial sharp drop in CV costs starts to level out. In our case, the right size of the tree is 3. The corresponding cost complexity parameter \( k \) is 2.0 as shown in the top scale of the graph.

**Pruning**

Now, that we know the cost complexity parameter for the right size of the tree, we can snip off the least important splits and get the final classification tree. The function \texttt{rprune} does the job:

\begin{verbatim}
[0] z=rprune k
[1] k=k+k
[2] rexx'tpr<-prune.tree(t,method= "misclass",k=1,k,')
[3] rexx'tm<-as.matrix(tpr[[1]])'
[4] rexx'tm<-cbind(rownames(tm),tm)'
[5] z=rget'tm'
\end{verbatim}

The only argument of the function is the value of the cost complexity parameter. In line [1] we convert it to a character vector to compose a command to R (line
The meaning of the other lines is the same as in the function `rtree`.

To simplify the code we used constant names for the pruned tree (`tpr`) and the matrix of the tree structure (`tm`).

Let us apply `rprune` to our tree and plot the right-sized classification tree (Figure 6):

```apl
i+×+row[1]×2
l+i+t[;1;i]
row←t[;1]
:EndWhile
z+→row[4]
```

The left argument is a new observation to be classified. The right argument is the tree matrix. To classify a new object we only need the 1st, 2nd, 5th and 6th columns of the tree matrix so in line [1] we form a new matrix containing those columns only. As all the elements of the tree matrix are nested character vectors and we need the first column to be numeric to easily navigate through the tree we convert the column to numeric type (line [2]). The procedure of classifying a new object is based on “dropping” the object through the tree till the current node is terminal. This is done in the while loop (lines [4-14]). As it was said, the predictors can be of either numeric or factor scale, so in line [6] we check if the current predictor is numeric. This is done by checking if the first element of the threshold vector is ‘<’. This element is ‘:’ for factor predictors. Then we check if the logical rule for the current node is true or false (line [7] for numeric predictors and line [9] for factors) and find the next node to jump to (lines [11-13]). When a terminal node is reached, the 4th element of the corresponding node contains the name of the class for the new object (line [15]).

The following examples illustrate the use of `predict`. First, let’s apply the function to the maximal tree `t`:

```apl
iris[1;;1] predict t
Setosa
iris[51;;1] predict t
Versicolor
iris[101;;1] predict t
Virginica
```

Just as expected, the tree `t` classifies all objects of the learning data set with no errors. Let’s examine the classification tree of the right size:

```apl
pr2+←(¢[2]iris)predict"<t2
+/>pr2="iris[;1;1]
```

8 APL Quote Quad
So, there are 6 misclassified objects or, in other words, the prediction error is only 4%. At the same time the complexity (the size of the tree) of a tree-based model decreased from 9 to 3 nodes!

**Speed of computation**

As it was said above, one of the main reasons to use the R-APL interface is the speed of calculations. Let us now compare the time required to grow a classification tree of the maximal size for the cover function `rtree` that calls the R function `tree()` and for the APL function `GrowTree` described in [4]. Let us first prepare an artificial data set of random values:

```r
data<-4000.30p(1200/1200)
p<data[;30]
```

Since the APL function `GrowTree` expects the response variable to be in the last column, there are 40 classes in the data set. Now we can measure the time required to build a classification tree:

```r
t1<tts`tr.GrowTree data=t1~ts-t1 t1 0 0 0 0 0 25 0
```

So, it takes `GrowTree` about 25 seconds to grow a classification tree.

To simplify the code for `rtree`, the response variable must be in the first column and we have to explicitly make it nested or else we will grow a regression rather than a classification tree (this can also be done by adding the command `re<`data[,1]<as.factor(data[,1])` to the function `rtree`):

```r
data2<~t+{}+{} data2\{1\}+{} data2\{1\} =data2 2 p<data2\{1\} 40 t2<~ts=tr tree data2=t2~ts-t2 t2 0 0 0 0 0 6 0
```

So, R-based calculations are more than 4 times faster than that of APL functions in this case! The difference certainly depends on the number of factors such as the RAM available, processor speed and others but as the data sets get larger this difference gets more noticeable.

**Conclusion**

In this paper we have shown the basic steps to build a working interface between the R statistical system and the Dyalog APL workspace. A few examples of passing simple vectors and arrays to the R session, submitting R commands, calling R functions and getting the results back to APL have been demonstrated.

The examples dedicated to graphics illustrate the possibility of using a large set of R graphical techniques including simple X-Y plots, specific plots such as Box-and-Whisker plot and complex 3D plots.

Finally, a complex example of growing classification trees has been shown. The example includes calling R functions to grow, cross-validate, prune and plot classification trees for APL data. It has also been shown how nested arrays can be passed to the R session as well as how one can access complex R objects.

The paper can serve either as a short tutorial to the R-APL interface or as a base to create more complex programs for the joint use of the R statistical system and the APL programming language.

We sincerely hope the readers will find this paper interesting and helpful.

**References**


