R News

The Newsletter of the R Project

Volume 2/2, June 2002

Editorial

by Kurt Hornik

Welcome to this first regular issue of *R News* in 2002, following the special issue on applications of R in medical statistics. In the future, it is planned to have two non-patch releases of R per year around the beginning of April and October, and to have regular issues of *R News* within 6 weeks after these releases. This also means that R fans should have enough reading material for their summer and/or winter breaks.

As the R Project keeps gaining in size and complexity, it needs to gradually replace its traditional approach of collectively managing all R tasks by focusing responsibilities more explicitly. To this end, the Editorial Board of *R News* has been reduced from 'all of R Core' to three members: Fritz Leisch, Thomas Lumley, and myself. From now on, *R News* editors will each serve a three year period, acting as Editor-in-Chief in the final year. In 2003, Doug Bates will join the Editorial Board, with Fritz Leisch taking over as Editor-in-Chief and me stepping down. Bill Venables continues to serve as column editor for the Programmer's Niche. Suggestions and volunteers for additional columns are most welcome (Book reviews anyone? Of course, nowadays 'book' is a fairly general concept …). In addition, we welcome suggestions for special issues, and in particular volunteers for guest-editing them.

R 1.5.0 was released on 2002-04-29, with the main innovations in add-on packages (see "Changes in R" for detailed release information). Two articles by Brian Ripley and David Meyer describe enhancements in the standard time series package **ts**: much improved handling of missing values, full ARIMA support, stuctural equation modeling, and exponentially weighted smoothing and forecasting. In addition, there are now two more recommended packages, **grid** and **lattice**, implementing the 'new R graphics engine' and the R version of Trellis. These packages are introduced in articles by their respective authors, Paul Murrell and Deepayan Sarkar.

This issue has much more exciting information, including news on R and spatial statistics, distributed computing, and bioinformatics. We are sure it contains something for everyone.

Kurt Hornik
Wirtschaftsuniversität Wien, Austria
Technische Universität Wien, Austria
Kurt.Hornik@R-project.org

Contents of this issue:

Time Series in R 1.5.0

by Brian D. Ripley

R has shipped with a package **ts** since 0.65.0 in mid 1999; this package has been enhanced considerably in version 1.5.0. The original release brought together several existing functions written by Paul Gilbert, Martyn Plummer, Adrian Trapletti and myself and included interfaces to a Fortran package I wrote for teaching time series in the late 1970's. Improvements were promised at the time, one function being called `arima0` to signal its preliminary status.

There are contributed packages with other time series functionality, including **tseries** which has an econometric/financial slant, and bundle **dse**, a toolkit for working with state-space models of vector time series.

The new function `HoltWinters` is discussed in the following article by David Meyer.

One of the goals of package **ts** was to provide enough functionality to cover the time series chapter of Venables and Ripley (1999). Time-series analysis was part of the 'PLUS' of S-PLUS more than a decade ago, but unfortunately has been little updated since, and I had planned for a long time to provide better facilities to fit ARIMA processes. Indeed, this was on the list of new things we intended to cover in Venables and Ripley (1999), and it would have been embarrassing not to have done so by Venables and Ripley (2002). So one of the innovations for R 1.5.0 is function `arima`, a version of which appears in the **MASS** library section for S-PLUS.

Updating package **ts** took so long because of technical challenges which this article hopes to illuminate. There can be a lot more detail to statistical computing than one finds even in the most complete monographs. It is often said that the way to learn a subject is to be forced to teach a course on it: writing a package is a much more searching way to discover if one really understands a piece of statistical methodology!

Missing values

Missing values are an occupational hazard in some applications of time series: markets are closed for the day, the rain gauge jammed, a data point is lost, ... So it is very convenient if the time-series analysis software can handle missing values transparently. Prior to version 1.5.0, R's functions made little attempt to do so, but we did provide functions `na.contiguous` and `na.omit.ts` to extract non-missing stretches of a time series.

There were several technical problems in supporting arbitrary patterns of missing values:

- Most of the computations were done in Fortran, and R's API for missing values only covers C code: this was resolved by carefully translating code[1] to C.

- Some computations only have their standard statistical properties if the series is complete. The sample autocorrelation function as returned by `acf` is a valid autocorrelation for a stationary time series if the series is complete, but this is not necessarily true if correlations are only computed over non-missing pairs. Indeed, it is quite possible that the pattern of missingness may make missing one or both of *all* the pairs at certain lags. Even if most of the pairs are missing the sampling properties of the ACF will be affected, including the confidence limits which are plotted by default by `plot.acf`.

- Standard time-series operations propagate missingness. This is true of most filtering operations, in particular the differencing operations which form the basis of the 'Box–Jenkins' methodology. The archetypal Box–Jenkins 'airline' model[2] involves both differencing and seasonal differencing. For such a model one missing value in the original series creates three missing values in the transformed series.

Our approach has been to implement support for missing values where we know of a reasonably sound statistical approach, but expect the user to be aware of the pitfalls.

Presidents

One of R's original datasets is `presidents`, a quarterly time series of the Gallup polls of the approval rating of the US presidents from 1945 to 1974. It has 6 missing values, one at the beginning as well the last two quarters in each of 1948 and 1974. This seems a sufficiently complete series to allow a reasonable analysis, so we can look at the ACF and PACF (figure 1)

```
data(presidents)
acf(presidents, na.action = na.pass)
pacf(presidents, na.action = na.pass)
```

[1]The usual route to translate Fortran code is to `f2c -a -A` from http://www.netlib.org/f2c. Unfortunately this can generate illegal C, as happened here, and if speed is important idiomatic C can run substantially faster. So the automatic translation was re-written by hand.

[2]as fitted to a monthly series of numbers of international airline passengers, now available as dataset `AirPassengers` in package **ts**.

We need an `na.action` argument, as the default behaviour is to fail if missing values are present. Function `na.pass` returns its input unchanged.

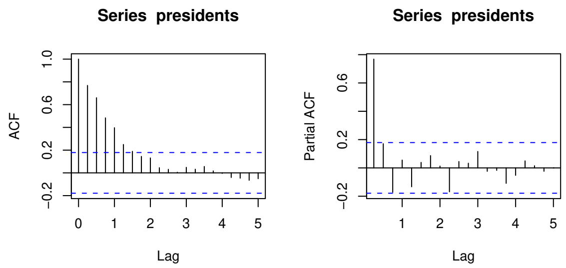


Figure 1: Autocorrelation and partial correlation plots of the `presidents` series. Note that the confidence limits shown do not take account of missingness.

The plots suggest an AR(1) or AR(3) model would be appropriate. We cannot use `ar`, but the new function `arima` will fit these models.

```
> (fit1 <- arima(presidents, c(1, 0, 0)))

Coefficients:
         ar1  intercept
      0.8242    56.1505
s.e.  0.0555     4.6434

sigma^2 estimated as 85.47:
log likelihood = -416.89,  aic = 839.78
> tsdiag(fit1)
> (fit3 <- arima(presidents, c(3, 0, 0)))

Coefficients:
         ar1     ar2      ar3  intercept
      0.7496  0.2523  -0.1890    56.2223
s.e.  0.0936  0.1140   0.0946     4.2845

sigma^2 estimated as 81.12:
log likelihood = -414.08,  aic = 838.16
> tsdiag(fit3)
```

This suggests a fairly clear preference for AR(3). Function `tsdiag` is a new generic function for diagnostic plots.

Fitting ARIMA models

There are several approximations to full maximum-likelihood fitting of ARIMA models in common use. For an ARMA model (with no differencing) the model implies a multivariate normal distribution for the observed series. A very common approximation is to ignore the determinant in the normal density. The 'conditional sum of squares' approach uses a likelihood conditional on the first few observations, and reduces the problem to minimizing a sum of squared residuals. The CSS approach has the same asymptotics as the full ML approach, and the textbooks often regard it as sufficient. It is not adequate for two reasons:

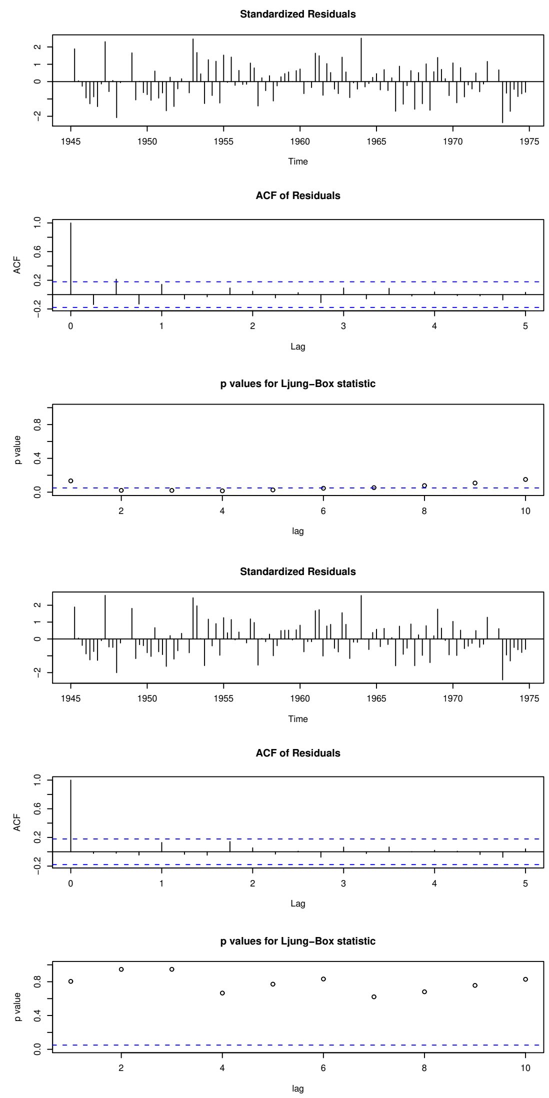


Figure 2: Diagnostic plots for AR(1) (upper) and AR(3) (lower) fits to the `presidents` series.

1. **Missing values.** The residuals are found by a recursive filter which cannot be adapted to handle missing values. The theory can be, but loses its main appeal: simplicity.

2. **Short series.** Both conditioning and discarding the determinant term are essentially end corrections. Unlike spatial statistics, edge effects are normally ignored in time series, in the mistaken belief that time series are long. For example, `AirPassengers` is of length 144: surely that is not short? Yes it is: we are fitting 12 linked series of length 12 each, one for each month, and the relevant end effect is that there are only 12 years. For pedagogical purposes `arima` and `arima0` include `method = "CSS"` so users can see for themselves the size of the effect.

The state-space approach ('Kalman filtering') has long been advocated as a good way to compute the likelihood of an ARMA model even in the presence of missing values: computational details are given by Durbin and Koopman (2001), and the original `arima0` used Fortran code[3] published as long ago as Gardner et al.. The state-space approach used to be regarded (rightly) as slow, but that is no longer an issue for tasks likely to be done in R. It is important to use state-space models correctly, including initializing them at their stationary distribution or the end effects will return.

ARIMA models present a further challenge. The model is for a differenced version of the observed data rather than for the data themselves, so a likelihood is not actually defined. There are a number of approaches outlined in Durbin and Koopman (2001),[4] of which I find the most satisfactory is the 'diffuse prior' approach, and that is implemented in `arima`. This assumes that the initial values of the time series on which the traditional approach conditions have mean zero (as someone chose the units of observation) but a large variance. Again there are both theoretical and numerical issues, as well as practical ones: what should one do if every January observation is missing? The approach in `arima` can cope with missing values in the initial segment.

Even when one has defined a log-likelihood and found an algorithm to compute it, there remains the task of optimizing it. Yet again this is much harder than the books make out. Careful study shows that there are often multiple local maxima. Finding good starting values is nigh to impossible if the end effects are going to be important. Finding reliable test examples is difficult. In the early days of `arima0` someone reported that it gave different results from SPSS, and Bill Venables suggested on `R-help` that this might prove helpful to the SPSS developers! One good test is to reverse the series and see if the same model is fitted. All ARIMA processes are reversible and `help("AirPassengers")` provides an empirical demonstration. So if `arima` takes a long time to find a solution, please bear in mind that a reliable solution is worth waiting for.

UK lung deaths

Venables and Ripley (2002, §14.3) analyse a time series from Diggle (1990) on monthly deaths from bronchitis, emphysema and asthma in the UK, 1974–1979. As this has only 6 whole years, it is a fairly severe test of the ability to handle end effects.

Looking at ACFs of the seasonally differenced series suggests an ARIMA($(2,0,0) \times (0,1,0)_{12}$) model, which we can fit by `arima`:

```
data(deaths, package="MASS")
deaths.diff <- diff(deaths, 12)
## plots not shown here
acf(deaths.diff, 30); pacf(deaths.diff, 30)

> (deaths.arima1 <-
    arima(deaths, order = c(2,0,0),
          seasonal = list(order = c(0,1,0),
                          period = 12)) )
Coefficients:
        ar1     ar2
      0.118  -0.300
s.e.  0.126   0.125

sigma^2 = 118960:
log likelihood = -435.83,  aic = 877.66
> tsdiag(deaths.arima1, gof.lag = 30)
```

However the diagnostics indicate the need for a seasonal AR term. We can try this first without differencing

```
> (deaths.arima2 <-
    arima(deaths, order = c(2,0,0),
          list(order = c(1,0,0),
               period = 12)) )
Coefficients:
        ar1     ar2   sar1  intercept
      0.801  -0.231  0.361    2062.45
s.e.  0.446   0.252  0.426     133.90

sigma^2 = 116053:
log likelihood = -523.16,  aic = 1056.3
> tsdiag(deaths.arima2, gof.lag = 30)
```

The AICs are not comparable, as a differenced model is not an explanation of all the observations.

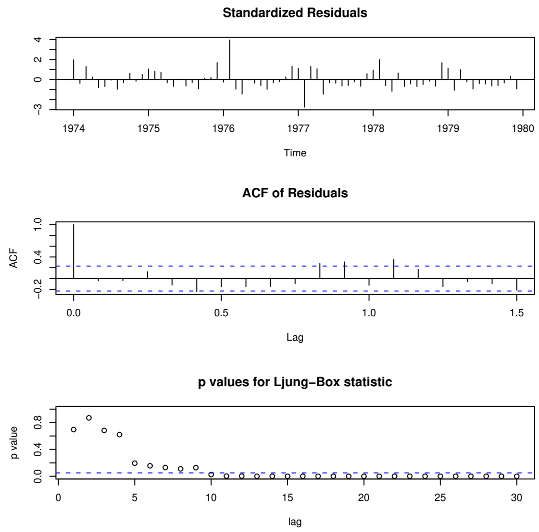


Figure 3: Diagnostic plots for `deaths.arima2`.

The diagnostics suggest that there is still seasonal structure in the residuals, so we next tried including both differencing and a seasonal AR term:

[3] Not only did this have to be translated to C to handle missing values, but several undocumented efficiency gains that resulted from assuming complete observation had to be undone.

[4] which in parts is an advertisement for non-free C code called **SsfPack** at `http://www.ssfpack.com` which links to a non-free Windows program called **Ox**.

```
> (deaths.arima3 <-
    arima(deaths, order = c(2,0,0),
          list(order = c(1,1,0),
               period = 12)) )
Coefficients:
       ar1     ar2    sar1
     0.293  -0.271  -0.571
s.e. 0.137   0.141   0.103

sigma^2 = 77145:
log likelihood = -425.22,  aic = 858.43
> tsdiag(deaths.arima3, gof.lag = 30)
```

for which the diagnostics plots look good.

Structural time series

Structural models of time series are an approach which is most closely linked to the group of Andrew Harvey (see in particular Harvey, 1989) but there are several closely related approaches, such as the dynamic linear models of Jeff Harrison and co-workers see (West and Harrison (1997); Pole et al. (1994) is a gentler introduction).

I have long thought that the approach had considerable merit, but believe its impact has been severely hampered by the lack of freely-available software.[5] It is beginning to appear in introductory time-series textbooks, for example Brockwell and Davis (1996, §8.5). As often happens, I wrote `StructTS` both for my own education and to help promote the methodology.

Experts such as Jim Durbin have long pointed out that all the traditional time-series models are just ways to parametrize the second-order properties of stationary time series (perhaps after filtering/differencing), and even AR models are not models of the real underlying mechanisms. (The new functions `ARMAacf`, `ARMAtoMA` and `acf2AR` allow one to move between the ACF representing the second-order properties and various parametrizations.) Structural time series are an attempt to model a plausible underlying mechanism for non-stationary time series.

The simplest structural model is a *local level*, which has an underlying level m_t which evolves by

$$\mu_{t+1} = \mu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)$$

The observations are

$$x_t = \mu_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

so there are two parameters, σ_ξ^2 and σ_ϵ^2, either of which could be zero. It is an ARIMA(0,1,1) model, but with restrictions on the parameter set.

The next step up is *local linear trend model* which has the same measurement equation, but with a time-varying slope in the dynamics for μ_t, given by

$$\begin{aligned}\mu_{t+1} &= \mu_t + \nu_t + \xi_t, \qquad \xi_t \sim N(0, \sigma_\xi^2)\\ \nu_{t+1} &= \nu_t + \zeta_t, \qquad \zeta_t \sim N(0, \sigma_\zeta^2)\end{aligned}$$

with three variance parameters. It is not uncommon to find $\sigma_\zeta^2 = 0$ (which reduces to the local level model) or $\sigma_\xi^2 = 0$, which ensures a smooth trend. This is a (highly) restricted ARIMA(0,2,2) model.

For a seasonal model we will normally use the so-called *Basic Structural Model* (BSM) for a seasonal time series. To be concrete, consider energy consumption measures quarterly, such as the series `UKgas` in package **ts**. This is based on a (hypothetical) decomposition of the series into a level, trend and a seasonal component. The measurement equation is

$$x_t = \mu_t + \gamma_t + \epsilon_t, \qquad \epsilon_t \sim N(0, \sigma_\epsilon^2)$$

where γ_t is a seasonal component with dynamics

$$\gamma_{t+1} = -(\gamma_t + \gamma_{t-1} + \gamma_{t-2}) + \omega_t, \quad \omega_t \sim N(0, \sigma_\omega^2)$$

The boundary case $\sigma_\omega^2 = 0$ corresponds to a deterministic (but arbitrary) seasonal pattern. (This is sometimes known as the 'dummy variable' version of the BSM.) There are now four variance parameters $(\sigma_\zeta^2, \sigma_\xi^2, \sigma_\omega^2, \sigma_\epsilon^2)$, one or more of which (but not all) can be zero.

These models are quite similar to those used in exponential smoothing and Holt–Winters forecasting (see the accompanying article by David Meyer); one important difference is that structural time series are handled in a formal statistical framework.

Estimation

Structural time series are fitted by maximum likelihood with a diffuse prior for those components (such as the overall level) which are not fully specified by the model. Using the state-space approach (as detailed in Durbin and Koopman, 2001) makes it easy to handle missing values.

What is not easy is the optimization, and I am now convinced that many of the published examples (e.g., Figure 8-4 in Brockwell and Davis (1996)) are incorrect. One issue is multiple local maxima of the log-likelihood, and some examples do seem to be sub-optimal local maxima. Optimization under non-negativity constraints can be tricky,[6] and it appears that other software either assumes that all the variances are positive or that $\sigma_\epsilon^2 > 0$.

We can illustrate this on the classic airline passengers dataset.

[5] Harvey's coworkers have produced a Windows package called STAMP, http://stamp-software.com, and **SsfPack** provides code in **Ox** for these models.

[6] Function `StructTS` uses `optim(method="L-BFGS-B")`, and it was this application that I had in mind when programming that.

```
data(AirPassengers)
## choose some sensible units on log scale
ap <- log10(AirPassengers) - 2
(fit <- StructTS(ap, type= "BSM"))
Call:
StructTS(x = ap, type = "BSM")

Variances:
   level     slope      seas   epsilon
0.000146  0.000000  0.000263  0.000000
```

Note that the measurement variance σ_ϵ^2 is estimated as zero.

Prediction and smoothing

One we have fitted a structural time series model, we can compute predictions of the components, that is the level, trend and seasonal pattern. There are two distinct bases on which we can do so. The `fitted` method for class `"StructTS"` displays the contemporaneous predictions, whereas the `tsSmooth` method displays the final predictions. That is, `fitted` shows the predictions of μ_t, ν_t and γ_t based on observing $x_1, \ldots, x_t$ for $t = 1, \ldots, T$, whereas `tsSmooth` gives predictions of μ_t, ν_t and γ_t based on observing the whole time series $x_1, \ldots, x_T$ (the 'fixed interval smoother' in Kalman filter parlance).

Looking at the fitted values shows how the information about the trend and seasonal patterns (in particular) builds up: it is like preliminary estimates of inflation etc given out by national statistical services. The smoothed values are the final revisions after all the information has been received and incorporated.

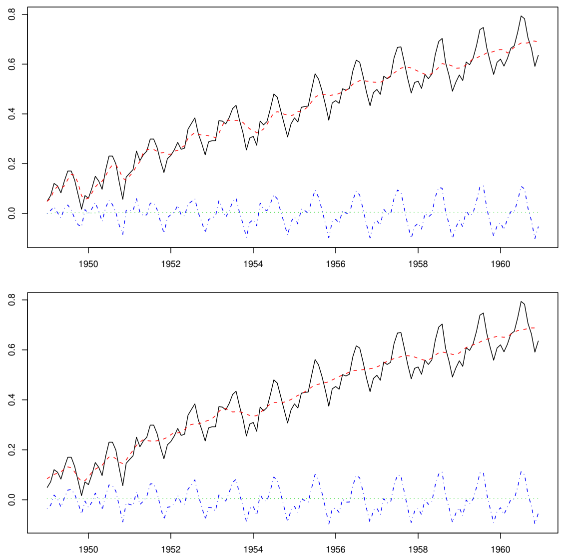


Figure 4: Fitted values (top) and smoothed values (bottom) of a BSM fit to the `AirPassengers` data, on $\log_{10}$ scale. The original series is solid black, the level is dashed red, the slope is dotted green and the seasonal component is dash-dotted blue.

We can illustrate this for the `AirPassengers` data by

```
plot(cbind(ap, fitted(fit)),
   plot.type = "single", lty=1:4, col=1:4)
plot(cbind(ap, tsSmooth(fit)),
   plot.type = "single", lty=1:4, col=1:4)
```

Note the way that fluctuations which are contemporaneously ascribed to level changes are revised to be part of the (changing) seasonal component as the latter changes shape around 1952–4. It is instructive to compare this with Figure 8-4 in Brockwell and Davis (1996).

Implementation

The tools available to the R programmer have moved on considerably since 1999. Like much of the statistical functionality in R, the internals of package **ts** were written in C or Fortran, interfaced via `.C` or `.Fortran`. Nowadays much new code is being written using C code interfaced with `.Call` which allows the R objects to be manipulated at C level, and in particular for the return object to be constructed in C code. The approach via `.Call` can also be much more efficient as it allows the programmer much more control over the copying of objects.

The first version of `arima` was written using the general-purpose Kalman filter code in the (new) functions `KalmanLike` and friends. This proved to be far too slow for seasonal ARIMA models, the time being spent in multiplying sparse matrices (and this was confirmed by profiling). The code was supplanted by special-purpose code, first for ARMA and then for ARIMA processes: having the slower reference implementation was very useful.

The original `arima0` code had dynamically allocated global C arrays to store information passed down and retrieved by `.C` calls. This had been flagged as a problem in the planned move to a threaded version of R. A more elegant approach was needed. `arima` passes around an R list which represents the state space model, but `arima0` only works at C level, so the obvious approach is to store all the information (which includes the data) in a C structure rather than in global variables. The remaining issue was to associate the instance of the structure with the instance of `arima0` (since in a threaded version of R more than one could be running simultaneously or at least interleaved). This was solved by the use of *external references*. This is a little-documented area of R (but see `http://developer.r-project.org/simpleref.html` and `http://developer.r-project.org/references.html`) that allows us to return *via* `.Call` an R object (of type `EXTPTRSXP`) that contains a pointer to our structure. We chose to de-allocate the arrays in the structure in the `on.exit` function of `arima0`, but it is now possible to get the R garbage collector to do this via a *finalizer*.

Bibliography

Peter J. Brockwell and Richard A. Davis. *Introduction to Time Series and Forecasting*. Springer-Verlag, New York, 1996. 5, 6

P. J. Diggle. *Time Series: A Biostatistical Introduction*. Oxford University Press, Oxford, 1990. 4

J. Durbin and S. J. Koopman, editors. *Time Series Analysis by State Space Methods*. Oxford University Press, Oxford, 2001. ISBN 0-19-852354-8. 4, 5

G. Gardner, A. C. Harvey, and G. D. A. Phillips. Algorithm as154. an algorithm for exact maximum likelihood estimation of autoregressive-moving average models by means of kalman filtering. *Applied Statistics*, 29:311–322. 4

A. C. Harvey. *Forecasting, Structural Time Series Models and the Kalman Filter*. Cambridge University Press, 1989. 5

A. Pole, M. West, and J. Harrison. *Applied Bayesian Forecasting and Time Series Analysis*. Chapman & Hall, 1994. 5

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S-PLUS*. Springer-Verlag, New York, third edition, 1999. 2

W. N. Venables and B. D. Ripley. *Modern Applied Statistics with S*. Springer-Verlag, New York, fourth edition, 2002. 2, 4

Mike West and Jeff Harrison. *Bayesian Forecasting and Dynamic Models*. Springer-Verlag, New York, second edition, 1997. ISBN 0-387-94725-6. 5

Brian D. Ripley
University of Oxford, UK
ripley@stats.ox.ac.uk

Naive Time Series Forecasting Methods

The Holt-Winters Method in package ts

by David Meyer

Exponential smoothing methods forecast time series by discounted past observations. They have become very popular because of their (relative) simplicity compared to their good overall performance. Common applications range from business tasks (e.g., forecasting of sales or stock fluctuations) to environmental studies (e.g., measurements of atmospheric components or rainfall data)—with typically no more *a priori* knowledge than the possible existence of trend of seasonal patterns. Such methods are sometimes also called *naive* because no covariates are used in the models, i.e., the data are assumed to be self-explaining. Their success is rooted in the fact that they belong to a class of *local* models which automatically adapt their parameters to the data during the estimation procedure and therefore implicitly account for (slow) structural changes in the training data. Moreover, because the influence of new data is controlled by hyperparameters, the effect is a smoothing of the original time series.

Among the simplest methods is the ordinary exponential smoothing, which assumes no trend and no seasonality. Holt's linear trend method (see Holt, 1957) and Winters extensions (see Winters, 1960) add a trend and a seasonal component (the latter either additive or multiplicative). Their methods are still surprisingly popular, although many extensions and more general frameworks do exist. We describe briefly the methods implemented in package **ts** and give some examples of their application.

Exponential smoothing

Let Y_t denote a univariate time series. Exponential smoothing assumes that the forecast $\hat{Y}$ for period $t+h$ based on period t is given by a variable level $\hat{a}$ at period t

$$\hat{Y}_{t+h} = \hat{a}_t \tag{1}$$

which is recursively estimated by a weighted average of the observed and the predicted value for Y_t:

$$\begin{aligned} \hat{a}_t &= \alpha Y_t + (1-\alpha)\hat{Y}_t \\ &= \alpha Y_t + (1-\alpha)\,\hat{a}_{t-1} \end{aligned}$$

$0 < \alpha < 1$ called the smoothing parameter; the smaller it is chosen, the less sensitive $\hat{Y}$ becomes for changes (i.e., the smoother the forecast time series will be). The initial value $\hat{a}_1$ is usually chosen as Y_1.

An equivalent formulation for $\hat{a}_t$ is the so called error-correction form:

$$\begin{aligned} \hat{a}_t &= \hat{a}_{t-1} + \alpha\,(Y_t - \hat{a}_{t-1}) \\ &= \hat{a}_{t-1} + \alpha\,(Y_t - \hat{Y}_t) \\ &= \hat{a}_{t-1} + \alpha\, e_t \end{aligned} \tag{2}$$

showing that $\hat{a}_t$ can be estimated by $\hat{a}_{t-1}$ plus an error e made in period t. (We will come back to this later on.)

Exponential smoothing can be seen as a special case of the Holt-Winters method with no trend and no seasonal component. As an example, we use the

Nile data for which an intercept-only model seems appropriate[1]. We will try to predict the values from 1937 on, choosing $\alpha = 0.2$:

```
library(ts)
data(Nile)
past   <- window(Nile, end   = 1936)
future <- window(Nile, start = 1937)
alpha  <- 0.2
```

To obtain a level-only model, we set the other hyper-parameters β and γ (see the next sections) to 0:

```
model <- HoltWinters(past, alpha = alpha,
                     beta = 0, gamma = 0)
```

The object `model` contains an object of type `"HoltWinters"`. The fitted values (obtained by `fitted(model)`) should of course be identical[2] to those given by `filter`:

```
filter(alpha * past, filter = 1 - alpha,
       method = "recursive", init = past[1])
```

We predict 43 periods ahead (until the end of the series):

```
pred <- predict(model, n.ahead = 43)
```

The `plot` method for Holt-Winters objects shows the observed and fitted values, and optionally adds the predicted values (see figure 1):

```
plot(model, predicted.values = pred)
lines(future)
```

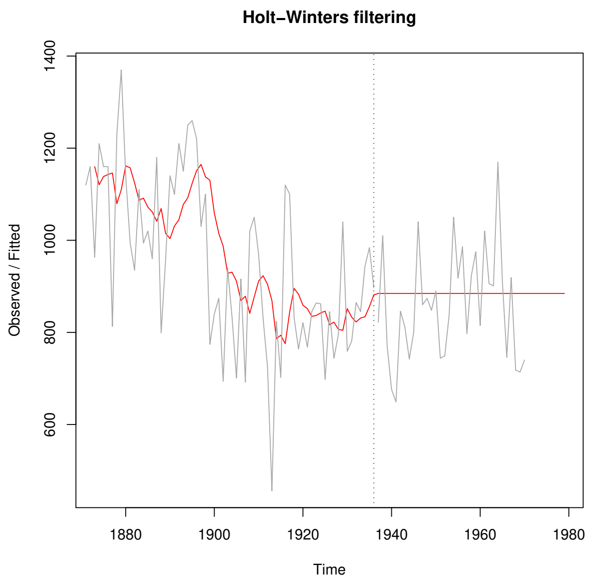


Figure 1: Exponential smoothing for the Nile data up to 1937, and the predictions in 1937 of the rest of the series. The original series is in shown in gray.

The coefficients (here only the level) are listed in the output produced by `print()` or can be extracted with `coef()`.

Holt's linear trend method

An obvious extension is to include an additional trend component, say $\hat{b}_t$. The forecast equation becomes:

$$\hat{Y}_{t+h} = \hat{a}_t + h \cdot \hat{b}_t$$

with updating formulas expressing similar ideas to those for exponential smoothing. There is an additional formula for the trend component:

$$\begin{aligned} \hat{a}_t &= \alpha Y_t + (1-\alpha)(\hat{a}_{t-1} + \hat{b}_{t-1}) \\ \hat{b}_t &= \beta\,(\hat{a}_t - \hat{a}_{t-1}) + (1-\beta)\,\hat{b}_{t-1} \end{aligned}$$

The use is a *local* fit of a straight line with the coefficients adjusted for new values. We now have one more parameter to choose: β. A straightforward approach to find the optimal values for both α and β is to look for the OLS estimates, i.e., the parameter combination minimizing the sum of squared errors of the one-step-ahead predictions. That is what `HoltWinters` does for all the unspecified parameters, illustrated below for the Australian residents data:

```
data(austres)
past   <- window(austres, start = c(1985, 1),
                          end = c(1989, 4))
future <- window(austres, start = c(1990, 1))
(model <- HoltWinters(past, gamma = 0))

Holt-Winters exponential smoothing with trend
and without seasonal component.

Call:
 HoltWinters(x = past, gamma = 0)

Smoothing parameters:
 alpha:  0.931416
 beta :  0.494141
 gamma:  0

Coefficients:
         [,1]
a 16956.68845
b    63.58486
```

The start values for $\hat{a}_t$ and $\hat{b}_t$ are $Y[2]$ and $Y[2] - Y[1]$, respectively; the parameters are estimated via `optim` using `method = "L-BFGS-B"`, restricting the parameters to the unit cube.

Another feature is the optional computation of confidence intervals for the predicted values, which by default are plotted along with them (see figure 2):

```
pred <- predict(model, n.ahead = 14,
                prediction.interval = TRUE)
plot(model, pred); lines(future)
```

[1] At least piecewise: note that one might expect a structural break in 1899 due to the construction of the first Ashwan dam

[2] Currently the initial value is only observation 2 as it is for the trend model (for which this is sensible). This will be changed in R 1.5.1.

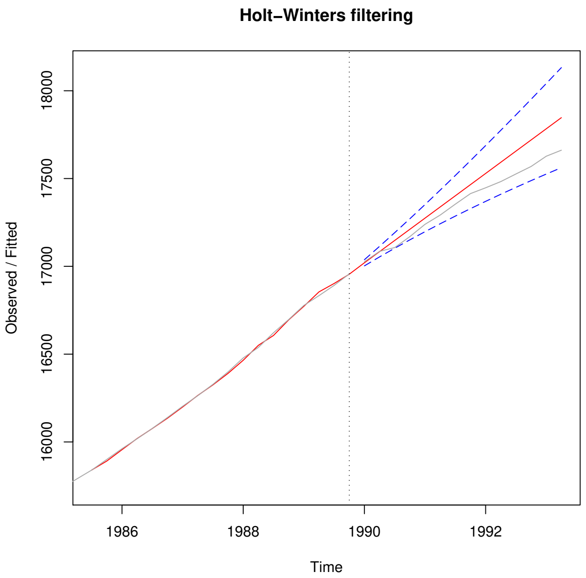


Figure 2: Holt's linear trend method applied to data on Autralian residents, with 95% confidence limits in dashed blue lines. Original data are plotted in gray, fitted and predicted values in red.

Winters' seasonal method

Winters extended the method to include a seasonal component. There are additive and multiplicative versions. In the following, we suppose seasonal trend components $\hat{s}_t$ and a period p (e.g., 4 for quarterly data).

Additive version

The additive model assumes the decomposition:

$$\hat{Y}_{t+h} = \hat{a}_t + h \cdot \hat{b}_t + \hat{s}_t$$

with an additional updating formula for the seasonal component:

$$\begin{aligned} \hat{a}_t &= \alpha\,(Y_t - \hat{s}_{t-p}) + (1-\alpha)(\hat{a}_{t-1} + \hat{b}_{t-1}) \\ \hat{b}_t &= \beta\,(\hat{a}_t - \hat{a}_{t-1}) + (1-\beta)\,\hat{b}_{t-1} \\ \hat{s}_t &= \gamma\,(Y_t - \hat{a}_t) + (1-\gamma)\,\hat{s}_{t-p} \end{aligned}$$

Multiplicative version

This version assumes a model with time-increasing seasonal component (i.e., the amplitude becomes larger with increasing t). The prediction formula becomes:

$$\hat{Y}_{t+h} = (\hat{a}_t + h \cdot \hat{b}_t)\,\hat{s}_t$$

and the updating formulas change accordingly:

$$\begin{aligned} \hat{a}_t &= \alpha\,(Y_t - \hat{s}_{t-p}) + (1-\alpha)(\hat{a}_{t-1} + \hat{b}_{t-1}) \\ \hat{b}_t &= \beta\,(\hat{a}_t - \hat{a}_{t-1}) + (1-\beta)\,\hat{b}_{t-1} \\ \hat{s}_t &= \gamma\,(Y_t / \hat{a}_t) + (1-\gamma)\,\hat{s}_{t-p} \end{aligned}$$

For automatic parameter selection, we need at least three complete cycles to estimate an initial seasonal component, done via a classical seasonal decomposition using moving averages performed by the new function `decompose()`. The intercept and trend are estimated by a simple regression on the extracted trend component.

As an example, we apply the multiplicative version to the UK gas consumption data for 1960–80 and predict the next six years.

```
data(UKgas)
past   <- window(UKgas, end   = c(1980, 4))
future <- window(UKgas, start = c(1981, 1))
model  <- HoltWinters(past, seasonal = "mult")
pred   <- predict(model, n.ahead = 24)
```

For clarity, we plot the original time series and the predicted values separately (see figure 3):

```
par(mfrow = c(2,1))
plot(UKgas, main = "Original Time Series")
plot(model, pred)
```

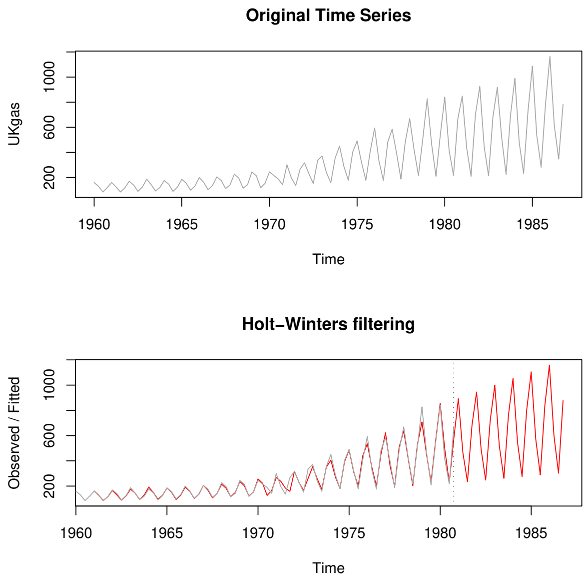


Figure 3: Winter's method applied to quarterly UK gas consumption data. The upper figure is the actual data, the lower the data (gray) and fited values (red) up to 1980Q4, the predictions (red) thereafter.

Conclusion

We have seen that the Holt-Winters method consists in a simple yet effective forecasting procedure, based on exponential moving averages, covering both trend and seasonal models. Many extensions exist however (see, e.g., Hyndman et al., 2001), and the various formulations can also be seen as special cases of a wider class of models. For example, all but the multiplicative Holt-Winters can be identified as (restricted) SARIMA models. To motivate this, consider equation 2—the error-correction form of exponential smoothing. By plugging it into the forecasting equation 1, we almost directly obtain:

$$\begin{aligned}(1-L)\hat{Y}_t &= \alpha\, e_{t-1} \\ &= (1-\theta L)e_t\end{aligned}$$

($\theta = 1 - \alpha$), which is (in terms of estimates) the standard form of an ARIMA(0,1,1)-process. Finally, we note that we actually face *state-space* models: given process Y_t, we try to estimate the underlying processes a_t, b_t and s_t which cannot be observed directly. But this is the story of structural time series and the function `structTS`, told in an another article by Brian D. Ripley ...

Bibliography

Holt, C. (1957). Forecasting seasonals and trends by exponentially weighted moving averages. *ONR Research Memorandum, Carnegie Institute 52*. 7

Hyndman, R., Koehler, A., Snyder, R., & Grose, S. (2001). A state space framework for automatic forecasting using exponential smoothing methods. *Journal of Forecasting*. To appear. 9

Winters, P. (1960). Forecasting sales by exponentially weighted moving averages. *Management Science*, **6**, 324–342. 7

David Meyer
Technische Universität Wien, Austria
David.Meyer@ci.tuwien.ac.at

Rmpi: Parallel Statistical Computing in R

by Hao Yu

Rmpi is an R interface (wrapper) to the Message-Passing Interface (MPI). MPI is a standard application interface (API) governed by the MPI forum (http://www.mpi-forum.org) for running parallel applications. Applicable computing environments range from dedicated Beowulf PC clusters to parallel supercomputers such as IBM's SP2. Performance and portability are the two main strengths of MPI. In this article, we demonstrate how the MPI API is implemented in **Rmpi**. In particular, we show how interactive R slaves are spawned and how we use them to do sophisticated MPI parallel programming beyond the "embarrassingly parallel".

Introduction

Put simply, parallel statistical computing means dividing a job into many small parts which will be executed simultaneously over a cluster of computers linked together in a network (LAN or WAN). Communications among the components is a crucial aspect of parallel computing. The message-passing model is highly suitable for such a task. There are two primary open-source message-passing models available: MPI and PVM (Parallel Virtual Machine, Geist et al., 1994). For PVM's implementation **rpvm** in R, see Li and Rossini, 2001.

Whether to use MPI or PVM largely depends on the underlying hardware structure and one's personal preference. Both models provide a parallel programming interface. The main difference is in how buffers are managed or used for sending and receiving data. Typically, in PVM, users are responsible for packing data into a system designated buffer on the sender side and unpacking from a system buffer on the receiver side. This reduces the user's responsibility for managing the buffer; on the other hand, this may create problems of running out of buffer space if large amounts of data are transferred. PVM can also lead to performance draining copies. MPI, by contrast, requires no buffering of data within the system. This allows MPI to be implemented on the widest range of platforms, with great efficiency. Perhaps Luke Tierney's package **snow** (Simple Network of Workstations, http://www.stat.umn.edu/~luke/R/cluster) will ultimately provide a unified interface to both **Rmpi** and **rpvm** so users can freely choose either one to implement parallel computing.

There are 128 routines in the MPI-1 standard and 287 functions in the combined MPI (MPI-1 and MPI-2) standard. MPI is rich in functionality; it is also capable of handling the diversity and complexity of today's high performance computers. Likely the environment with highest potential for MPI use is the Beowulf cluster, a high-performance massively parallel computer built primarily out of commodity hardware components.

One of the goals of **Rmpi** is to provide an extensive MPI API in the R environment. Using R as an interface, the user can either spawn C(++) or Fortran programs as children or join other MPI programs. Another goal is to provide an interactive R master and slave environment so MPI programming can be done completely in R. This was achieved by implementing a set of MPI API extensions specifically designed for the R or R slave environments.

Installation

Although MPI is a standard, its form does not necessitate implementation of all defined functions. There are at least ten freely available implementations of MPI, the most notable of which are MPICH (http://www.mcs.anl.gov/mpi/mpich) and LAM-MPI (http://www.lam-mpi.org). **Rmpi** uses LAM-MPI primarily because its implementation of the MPI API is more comprehensive than other implementations. However, **Rmpi** tries to use as few LAM specific functions as possible to avoid portability problems.

Before attempting installing LAM and **Rmpi**, a Beowulf cluster should be set up properly. See http://www.beowulf.org for details.

Installing LAM

The source code or pre-compiled Linux RPMs can be downloaded from the LAM web site. When attempting to install LAM at the system level, one should ensure that there is no other MPI, such as MPICH, already installed. Two MPIs cannot coexist at the system level. Once LAM is installed, edit the file 'lamb-host.def' either in the '/etc/lam' system directory or 'etc' in the user's home directory to add all hosts in a cluster. LAM uses `rsh` by default for remote execution. This can be changed to `ssh`. Check LAM and SSH documents for setting ssh and public key authentication to avoid passwords.

Use `lamboot` to boot LAM environment and run `lamexec C hostname` to see if all hosts respond. LAM provides a set of tools for host management and MPI program monitoring (`mpitask`).

Installing Rmpi

For LAM installed in '/usr' or '/usr/local' or for the Debian system with installed packages **lam3**, **lam3-dev**, and **lam-runtime**, just use

```
R CMD INSTALL Rmpi_version.tar.gz
```

For LAM installed in another location, please use

```
R CMD INSTALL Rmpi_version.tar.gz
  --configure-args=--with-mpi=/mpipath
```

Rmpi relies on Luke Tierney's package **serialize** for sending or receiving arbitrary R objects across the network and it must be installed. If you want to use any random number distributions in parallel environment, please install Michael Li's package **rsprng**: a wrapper to SPRNG (Scalable Parallel Random Number Generators).

A sample Rmpi session

```
{karl:10} lamboot
{karl:11} R
> library(Rmpi)
Rmpi version: 0.4-4
Rmpi is an interface (wrapper) to MPI APIs
with interactive R slave functionalities.
See 'library (help=Rmpi)' for details.
Loading required package: serialize
> mpi.spawn.Rslaves(nslaves=3)
3 slaves are spawned successfully. 0 failed.
master (rank 0,comm 1) of size 4 is running
    on: karl
slave1 (rank 1,comm 1) of size 4 is running
    on: karl
slave2 (rank 2,comm 1) of size 4 is running
    on: karl
slave3 (rank 3,comm 1) of size 4 is running
    on: karl4
> mpi.remote.exec(mean(rnorm(1000)))
           X1          X2          X3
1 -0.04475399 -0.04475399 -0.04475399
> mpi.bcast.cmd(mpi.init.sprng())
> mpi.init.sprng()
Loading required package: rsprng
> mpi.remote.exec(mean(rnorm(1000)))
            X1            X2          X3
1 -0.001203990 -0.0002667920 -0.04333435
> mpi.bcast.cmd(free.sprng())
> mpi.close.Rslaves()
[1] 1
> free.sprng()
> mpi.exit()
[1] "Detaching Rmpi. Rmpi cannot be used
    unless relaunching R."
> q()
{karl:12} lamhalt
```

MPI implementation in R

MPI uses a number of objects for message-passing. The most important one is a *communicator*. There are two types of communicators: *intracommunicator* and *intercommunicator*. A *communicator* (*comm*) usually refers to an *intracommunicator* which is associated with a group of members (CPU nodes). Most point-to-point and collective operations among the members must go through it. An *intercommunicator* is associated with two groups instead of members.

Rmpi defines several pointers in system memory to represent commonly used MPI objects. When loading **Rmpi**, an array of size 10 is allocated for *comm* objects, and arrays of size 1 are allocated for *status* and *info* objects. These objects are addressed using the R argument assignment. For example, `comm = 0` represents *comm* object 0 which by default is assigned to MPI_COMM_WORLD. MPI datatypes integer, double, and character are represented by `type=1`, `type=2`, and `type=3` respectively. They match R's integer, double, and character datatypes. Other types require serialization.

A general R object can be serialized to characters (`type=3`) before sending and unserialized after receiving. On a heterogeneous environment, MPI takes care of any necessary character conversion provided characters are represented by the same number of bytes on both sending and receiving systems.

Rmpi retains the original MPI C interface. A notable exception is the omission of message length because of R's way of determining length.

Regarding receive buffer preparation, one can use `integer(n)` (`double(n)`) for an integer buffer (a double buffer) of size `n`. However, R does not have a function for creating a character buffer (`character(1)` only creates an empty string). The function `string(n)` is supplied to create a character vector with one length `n` element. For example,

```
> string(2)
[1] "  "
```

`string` can be used as a receiver buffer for either a character vector of length 1 or a binary character vector generated by serialization.

In `.First.lib`, the LAM/MPI runtime environment is checked by a LAM command `lamnodes`. If it is not up, `lamboot` will be launched to boot a LAM session. After **Rmpi** is loaded, R becomes an MPI master with one member only (i.e., itself). Then it can use `mpi.comm.spawn` to spawn children. During spawning, an *intercommunicator* (default to *comm* 2) is created. The master (group 1) and children (group 2) should use intercomm merger so they will be in the same group for point-to-point or collective operations.

Interactive R slaves

In this section, we give details on how to spawn R slaves and communicate with them. The main function is `mpi.spawn.Rslaves`. This spawns a set of R slaves by using a shell program 'Rslaves.sh' in the Rmpi installation directory. The slaves will run on nodes specified by either LAM or a user. 'Rslave.sh' usually makes R run in BATCH mode so input (Rscript) and output (log) are required. Those log files are uniquely named after their host names, master process id, and master comm number. They can be very useful when debugging.

The default Rscript, 'slavedaemon.R' in the Rmpi installation directory, performs the necessary steps (intercomm merger) to establish communication with the master and runs in a while loop (waiting for an instruction from the master). When slaves are spawned, they form their own group accessed by *comm* 0. After intercomm merger, slaves use the default *comm* `.comm` (=1) to communicate with the master while the master uses the default *comm* 1 to communicate with the slaves. If necessary, the master can spawn additional sets of R slaves by using *comm* 3, 4, etc. (slaves still use `.comm=1`).

To determine the maximum number of slaves to be spawned, **Rmpi** provides a function `mpi.universe.size` to show the total nodes (CPUs) available in a LAM session. The master should not participate in numerical computation since the number of master and slaves may then exceed the number of nodes available.

In a heterogeneous environment, the user may wish to spawn slaves to specific nodes to achieve uniform CPU speed. For example,

```
> lamhosts()
 karl  karl karl4 karl4 karl5 karl5
    0     1     2     3     4     5
> mpi.spawn.Rslaves(hosts=c(3, 4), comm=3)
2 slaves are spawned successfully. 0 failed.
master (rank 0,comm 3) of size 3 is running
    on: karl
slave1 (rank 1,comm 1) of size 3 is running
    on: karl4
slave2 (rank 2,comm 1) of size 3 is running
    on: karl5
```

Once R slaves are spawned (assuming 3 slaves on *comm* 1 and 2 slaves on *comm* 3), the master can use `mpi.bcast.cmd` to send a command to be executed by all slaves. For example,

```
> mpi.bcast.cmd(print(date()), comm=1)
```

will let all slaves (associated with *comm* 1) execute the command `print(date())` and the results can be viewed by `tail.slave.log(comm=1)`. The command

```
> mpi.bcast.cmd(print(.Last.value), comm=1)
```

tells the slaves to put their last value into the log. If the executed results should be returned to master, one can use

```
> mpi.remote.exec(date(), comm=1)
```

We can think of `mpi.remote.exec` as a parallel `apply` function. It executes "embarrassingly parallel" computations (i.e., those where no node depends on any other). If the master writes a function intended to be executed by slaves, this function must be transferred to them first. One can use `mpi.bcast.Robj2slave` for that purpose. If each slave intends to do a different job, `mpi.comm.rank` can be used to identify itself:

```
> mpi.remote.exec(mpi.comm.rank(.comm),
    comm=1)
  X1 X2 X3
1  1  2  3
> mpi.remote.exec(1:mpi.comm.rank(.comm),
    comm=3)
$slave1
[1] 1
$slave2
[1] 1 2
```

Often the master will let all slaves execute a function while the argument values are on the master. Rather than relying on additional MPI call(s), one can pass the original arguments as in the following example.

```
> x <- 1:10
> mpi.remote.exec(mean, x, comm=1)
   X1  X2  X3
1 5.5 5.5 5.5
```

Here, the slaves execute `mean(x)` with x replaced by 1:10. Note that `mpi.remote.exec` allows only a small amount of data to be passed in this way.

Example

Rmpi comes with several demos. The following two functions are similar to the demo script 'slave2PI.R'.

```
slave2 <- function(n) {
  request <-1; job <-2
  anytag <- mpi.any.tag(); mypi <- 0
  while (1) {
    ## send master a request
    mpi.send(integer(1), type=1, dest=0,
      tag=request,comm=.comm)
    jobrange<-mpi.recv(integer(2), type=1,
      source=0, tag=anytag, comm=.comm)
    tag <- mpi.get.sourcetag()[2]
    if (tag==job) #do the computation
      mypi <- 4*sum(1/(1+((seq(jobrange[1],
        jobrange[2])-.5)/n)^2))/n + mypi
    else break #tag=0 means stop
  }
  mpi.reduce(mypi, comm=.comm)
}
master2PI <- function(n, maxjoblen, comm=1) {
  tsize <- mpi.comm.size(comm)
  if (tsize < 2)
    stop("Need at least 1 slave")
  ## send the function slave2 to all slaves
  mpi.bcast.Robj2slave(slave2, comm=comm)
  #let slave run the function slave2
  mpi.remote.exec(slave2, n=n, comm=comm,
        ret=FALSE)
  count <- 0; request <- 1; job <- 2
  anysrc <- mpi.any.source()
  while (count < n) {
    mpi.recv(integer(1), type=1,
      source=anysrc, tag=request, comm=comm)
    src <- mpi.get.sourcetag()[1]
    jobrange <- c(count+1,
            min(count+maxjoblen, n))
    mpi.send(as.integer(jobrange), type=1,
      dest=src, tag=job, comm=comm)
    count <- count+maxjoblen
  }
  ## tell slaves to stop with tag=0
  for (i in 1:(tsize-1)) {
    mpi.recv(integer(1), type=1,
      source=anysrc, tag=request, comm=comm)
    src <- mpi.get.sourcetag()[1]
    mpi.send(integer(1), type=1,
      dest=src, tag=0, comm=comm)
  }
  mpi.reduce(0, comm=comm)
}
```

The function slave2 is to be executed by all slaves and master2PI is to be run on the master. The computation is simply a numerical integration of $\int_0^1 1/(1+x^2)\,dx$. qA load balancing approach is used, namely, the computation is divided into more small jobs than there are slaves so that whenever a slave finishes a job, it can request a new one and continue. This approach is particularly useful in a heterogeneous environment where CPU's speeds differ.

Notice that the wild card `mpi.any.source` is used by the master in `mpi.recv` for first-come-first-served (FCFS) type of queue service. During computation, slaves keep their computed results until the last step `mpi.reduce` for global reduction. Since the master is a member, it must do the same by adding 0 to the final number. For 10 small jobs computed by 3 slaves, one can

```
> master2PI(10000, 1000, comm=1) - pi
[1] 8.333334e-10
```

By replacing the slaves' computation part, one can easily modify the above codes for other similar types of parallel computation.

On clusters with other MPIs

Several parallel computer vendors implement their MPI based on MPICH without spawning functionality. Can **Rmpi** still be used with interactive R slaves capability? This will not be an issue if one installs LAM at the user level. However, this practice is often not allowed for two reasons. First, vendors already optimize MPIs for their hardware to maximize network performance; LAM may not take advantage of this. Second, a job scheduler is often used to dispatch parallel jobs across nodes to achieve system load balancing. A user-dispatched parallel job will disrupt the load balance.

To run **Rmpi** in such clusters, several steps must be taken. The detailed steps are given in the 'README' file. Here we sketch these steps.

1. Modify and install **Rmpi** with the system supplied MPI;

2. Copy and rename the file 'Rprofile' in the Rmpi installation directory to the user's root directory as '.Rprofile';

3. Run `mpirun -np 5 R -save -q` to launch 1 master and 4 slaves with default *comm* 1.

On some clusters `mpirun` may dispatch the master to a remote node which effectively disables R interactive mode. **Rmpi** will still work in pseudo-interactive mode with command line editing disabled. One can run R in `R CMD BATCH` mode as `mpirun -np 5 R CMD BATCH` *Rin Rout*. Notice that only the master executes the Rscript *Rin*, exactly the same as in interactive mode.

Discussion

MPI has many routines. It is not currently feasible to implement them all in **Rmpi**. Some of the routines are not necessary. For example, many of the data management routines are not needed, since R has its own sophisticated data subsetting tools.

Virtual topology is a standard feature of MPI. Topologies provide a high-level method for managing CPU groups without dealing with them directly. The *Cartesian* topology implementation will be the target of future version of **Rmpi**.

MPI profiling is an interesting area that may enhance MPI programming in R. It remains to be seen if the MPE (Multi-Processing Environment) library can be implemented in **Rmpi** or whether it will best be implemented as a separate package.

Other exotic advanced features of MPI under consideration are Parallel I/O and Remote Memory Access (RMA) for one-sided communication.

With parallel programming, debugging remains a challenging research area. Deadlock (i.e., a situation arising when a failed task leaves a thread waiting) and race conditions (i.e., bugs caused by failing to account for dependence on the relative timing of events) are always issues regardless of whether one is working at a low level (C(++) or Fortran) or at a high level (R). The standard MPI references can provide help.

Acknowledgements

Rmpi is primarily implemented on the SASWulf Beowulf cluster funded by NSERC equipment grants. Support from NSERC is gratefully acknowledged.

Thanks to my colleagues Drs. John Braun and Duncan Murdoch for proofreading this article. Their effects have made this article more readable.

Bibliography

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Snuderam. *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing*. The MIT Press, Massachusetts, 1994.

W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI-The Complete Reference: Volume 2, MPI-2 Extensions*. The MIT Press, Massachusetts, 1998.

W. Gropp, E. Lusk, and A. Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, Massachusetts, 1999a.

W. Gropp, E. Lusk, and R. Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. The MIT Press, Massachusetts, 1999b.

Michael Na Li and A.J. Rossini. RPVM: Cluster statistical computing in R. *R News*, 1(3):4–7, September 2001. URL http://CRAN.R-project.org/doc/Rnews/.

M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra. *MPI-The Complete Reference: Volume 1, The MPI Core*. The MIT Press, Massachusetts, 1998.

Hao Yu
University of Western Ontario
hyu@stats.uwo.ca

The grid Graphics Package

by Paul Murrell

Introduction

The **grid** package provides an alternative set of graphics functions within R. This article focuses on **grid** as a drawing tool. For this purpose, **grid** provides two main services:

1. the production of low-level to medium-level graphical components such as lines, rectangles, data symbols, and axes.

2. sophisticated support for arranging graphical components.

The features of **grid** are demonstrated via several examples including code. Not all of the details of the code are explained in the text so a close consideration of the output and the code that produced it, plus reference to the on-line help for specific **grid** functions, may be required to gain a complete understanding.

The functions in **grid** do not provide complete high-level graphical components such as scatterplots or barplots. Instead, **grid** is designed to make it very easy to build such things from their basic components. This has three main aims:

1. The removal of some of the inconvenient constraints imposed by R's default graphical functions (e.g., the fact that you cannot draw anything other than text relative to the coordinate system of the margins of a plot).

2. The development of functions to produce high-level graphical components which would not be very easy to produce using R's default graphical functions (e.g., the **lattice** add-on package for R, which is described in a companion article in this issue).

3. The rapid development of novel graphical displays.

Drawing primitives

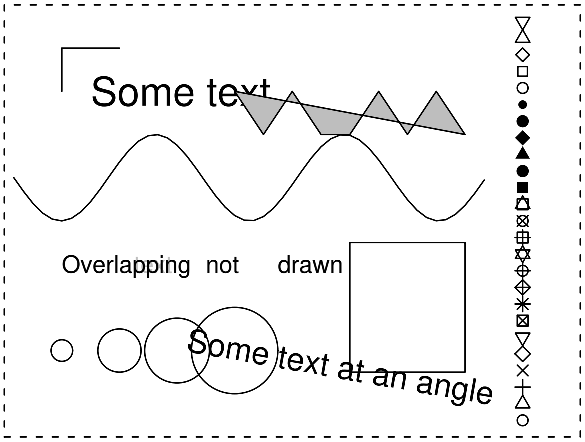


Figure 1: Example output from the **grid** primitive functions.

grid provides the standard set of basic graphical components: lines, text, rectangles, circles, polygons, and data symbols. A slight difference from the base graphics is that a set of text may be drawn such that any overlapping text is omitted. Also, **grid** has the notion of a "current location" and provides a command for resetting this location and a command for drawing a line from the previous location. The following set of code is from a sample **grid** session; the output produced by this code is shown in Figure 1.

```
grid.move.to(0.1, 0.8)
grid.line.to(0.1, 0.9)
grid.line.to(0.2, 0.9)
grid.text("Some text", x=0.15, y=0.8,
          just=c("left", "center"),
          gp=gpar(fontsize=20))
grid.text("Some text at an angle", x=0.85, y=0.1,
          just=c("right", "center"), rot=350,
          gp=gpar(fontsize=16))
grid.text(c("Overlapping", "text", "", "drawn"),
          0.1 + 0:3/8, 0.4, gp=gpar(col="grey"),
          just=c("left", "center"))
grid.text(c("Overlapping", "text", "not",
            "drawn"),
          0.1 + 0:3/8, 0.4,
          just=c("left", "center"),
          check.overlap=TRUE)
grid.circle(1:4/10, 0.2, r=1:4/40)
grid.points(rep(0.9, 25), 1:25/26, pch=1:25,
            size=unit(0.1, "inches"))
grid.polygon(0.4 + 0:8/20,
             0.6 + c(2,1,2,1,1,2,1,2,1)/10,
             gp=gpar(fill="grey"))
grid.rect(x=0.7, y=0.3, w=0.2, h=0.3)
grid.lines(x=1:50/60,
           y=0.6 + 0.1*sin(seq(-3*pi, 3*pi,
                                length=60)))
```

The functions are very similar to the base R counterparts, however, one important difference is in the way that graphical parameters, such as line colour and line thickness, are specified. There are default graphical parameter settings and any setting may be overridden by specifying a value via the `gp` argument, using the `gpar` function. There is a much smaller set of graphical parameters available:

`lty` line type (e.g., `"solid"` or `"dashed"`).

`lwd` line width.

`col` "foreground" colour for drawing borders.

`fill` "background" colour for filling shapes.

`font` a number indicating plain, bold, italic, or bold-italic.

`fontsize` the point size of the font.

`lineheight` the height of a line of text given as a multiple of the point size.

There may be additions to this list in future versions of **grid**, but it will remain much smaller than the list available in R's `par` command. Parameters such as `pch` are provided separately only where they are needed (e.g., in `grid.points()`).

Coordinate systems

All of the drawing in Figure 1 occurred within a so-called "normalised" coordinate system where the bottom-left corner of the device is represented by the location $(0,0)$ and the top-right corner is represented by the location $(1,1)$. For producing even simple statistical graphics, a surprising number of coordinate systems are required. **grid** provides a simple mechanism for specifying coordinate systems within rectangular regions, based on the notion of a *viewport*.

grid maintains a "stack" of viewports, which allows control over the context within which drawing occurs. There is always a default top-level viewport on the stack which provides the normalised coordinate system described above. The following commands specify a new coordinate system in which the y-axis scale is from -10 to 10 and the x-axis scale is from 0 to 5 (the call to `grid.newpage()` clears the device and resets the viewport stack):

```
grid.newpage()
push.viewport(viewport(yscale=c(-10, 10),
                       xscale=c(0, 5)))
```

All drawing operations occur within the context of the viewport at the top of the viewport stack (the *current viewport*). For example, the following command draws symbols relative to the new x- and y-scales:

```
grid.points(1:4, c(-3, 0, 3, 9))
```

In addition to x- and y-scales, a **grid** viewport can have a location and size, which position the viewport within the context of the previous viewport on the stack. For example, the following commands draw the points in a viewport that occupies only the central half of the device (the important bits are the specifications `width=0.5` and `height=0.5`):

```
grid.newpage()
push.viewport(
  viewport(width=0.5, height=0.5,
           yscale=c(-10, 10), xscale=c(0, 5)))
grid.points(1:4, c(-3, 0, 3, 9))
grid.rect(gp=gpar(lty="dashed"))
```

The margins around a plot in R are often specified in terms of lines of text. **grid** viewports provide a number of coordinate systems in addition to the normalised one and that defined by the x- and y-scales. When specifying the location and size of a graphical primitive or viewport, it is possible to select which coordinate system to use by specifying the location and/or size using a *unit* object. The following commands draw the points in a viewport with a margin given in lines of text. An x-axis and a y-axis, some labels, and a border are added to make something looking like a standard R plot (the output is shown in Figure 2). The important bits in the following code involve the use of the `unit()` function:

```
grid.newpage()
plot.vp <-
  viewport(x=unit(4, "lines"),
           y=unit(4, "lines"),
           width=unit(1, "npc") -
                 unit(4 + 2, "lines"),
           height=unit(1, "npc") -
                  unit(4 + 3, "lines"),
           just=c("left", "bottom"),
           yscale=c(-10.5, 10.5),
           xscale=c(-0.5, 5.5))
push.viewport(plot.vp)
grid.points(1:4, c(-3, 0, 3, 9))
grid.xaxis()
grid.text("X Variable",
          y=unit(-3, "lines"))
grid.yaxis()
grid.text("Y Variable",
          x=unit(-3, "lines"), rot=90)
grid.rect()
grid.text("Plot Title",
          y=unit(1, "npc") + unit(2, "lines"),
          gp=gpar(fontsize=14))
```

In terms of the arrangement of the graphical components, these few lines of code reproduce most of the layout of standard R plots. Another basic **grid** feature, *layouts*[1], allows a simple emulation of R's multiple rows and columns of plots.

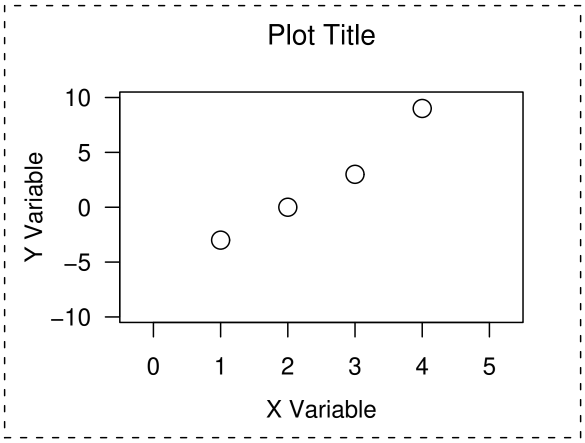


Figure 2: A standard scatterplot produced by **grid**.

If a viewport has a layout specified, then the next viewport in the stack can specify its position within that layout rather than using unit objects. For example, the following code draws the plot in the previous example within a 1-row by 2-column array (and an outer margin is added for good measure; the output is shown in Figure 3). The important bits to look for involve the specification of a layout for the first viewport, `layout=grid.layout(1, 2)`, and the specification of a position within that layout for the next viewport that is pushed onto the viewport stack, `layout.pos.col=1`:

```
grid.newpage()
push.viewport(
  viewport(x=unit(1, "lines"),
           y=unit(1, "lines"),
           width=unit(1, "npc") -
                 unit(2, "lines"),
           height=unit(1, "npc") -
                  unit(2, "lines"),
           just=c("left", "bottom"),
           layout=grid.layout(1, 2),
           gp=gpar(fontsize=6)))
grid.rect(gp=gpar(lty="dashed"))
push.viewport(viewport(layout.pos.col=1))
grid.rect(gp=gpar(lty="dashed"))
push.viewport(plot.vp)
grid.points(1:4, c(-3, 0, 3, 9))
grid.xaxis()
grid.text("X Variable", y=unit(-3, "lines"))
grid.yaxis()
grid.text("Y Variable", x=unit(-3, "lines"),
          rot=90)
grid.rect()
grid.text("Plot Title",
          y=unit(1, "npc") + unit(2, "lines"),
          gp=gpar(fontsize=8))
```

It should be noted that the `fontsize=6` setting in the first viewport overrides the default setting for all subsequent viewports in the stack and for all graphical components within the contexts that these view-

[1] These are similar to the facility provided by the base `layout` function, which mostly follows the description in Murrell, Paul R. (1999), but there is additional flexibility provided by the addition of extra units and these layouts can be nested by specifying multiple viewports in the viewport stack each with its own layout.

ports provide. This "inheritance" of graphical context from parent viewports is true for all graphical parameter settings.

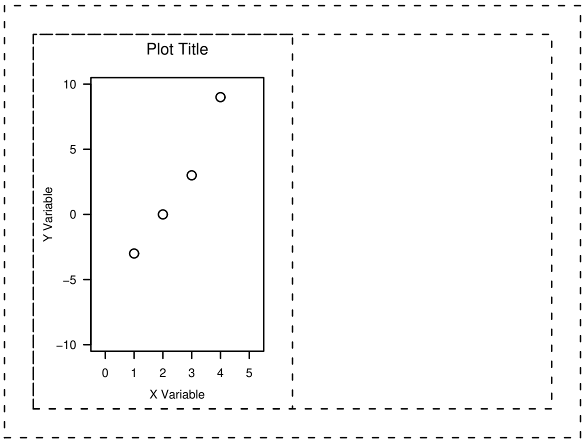


Figure 3: A **grid** scatterplot within a 1-by-2 array, within an outer margin.

There are three important features of this emulation of R's plot layout capabilities:

1. the control of the layout was performed completely within R code.
2. the amount of R code required was small.
3. the layout is actually more flexible than the R equivalent.

In order to illustrate the last point, the following examples add a couple of simple annotations to Figure 3 which are inconvenient or impossible to achieve in R.

The first three come from some requests to the R-help mailing list over over the last couple of years: drawing a plotting symbol that is a fixed number of millimeters square; drawing text within a plot in a location that is "in the top-left corner" rather than relative to the current axis scales; and drawing text at a location in the plot margins with an arbitrary rotation. The final example demonstrates the ability to draw from one coordinate system to another.

```
grid.rect(x=unit(1:4, "native"),
          width=unit(4, "mm"),
          y=unit(c(-3, 0, 3, 9), "native"),
          height=unit(4, "mm"))
grid.text("Text in\nthe upper-left\ncorner",
          x=unit(1, "mm"),
          y=unit(1, "npc") - unit(1, "mm"),
          just=c("left", "top"),
          gp=gpar(font=3))
grid.yaxis(main=FALSE, label=FALSE)
grid.text(c("-ten", "-five", "zero",
            "five", "ten"),
          x=unit(1, "npc") + unit(0.8, "lines"),
          y=unit(seq(-10, 10, 5), "native"),
          just=c("left", "centre"), rot=70)
pop.viewport(2)
push.viewport(viewport(layout.pos.col=2))
push.viewport(plot.vp)
grid.rect()
grid.points(1:4, c(-8, -3, -2, 4))

line.between <- function(x1, y1, x2, y2) {
  grid.move.to(unit(x2, "native"),
               unit(y2, "native"))
  pop.viewport(2)
  push.viewport(viewport(layout.pos.col=1))
  push.viewport(plot.vp)
  grid.line.to(unit(x1, "native"),
               unit(y1, "native"),
               gp=gpar(col="grey"))
  pop.viewport(2)
  push.viewport(viewport(layout.pos.col=2))
  push.viewport(plot.vp)
}
line.between(1, -3, 1, -8)
line.between(2, 0, 2, -3)
line.between(3, 3, 3, -2)
line.between(4, 9, 4, 4)
```

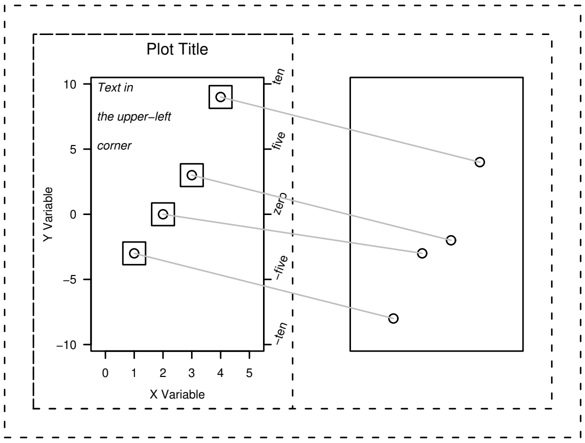


Figure 4: A **grid** scatterplot with simple annotations.

Applications of grid

R's base graphics are based on the notion of a plot which is surrounded by margins for axes and labels. Many statistical graphics cannot be conveniently described using such a plot as the basic building block. A good example, is the Trellis Graphics system (Becker et al., 1996; Cleveland, 1993), particularly the multipanel Trellis displays. Here, the more natural building block is a "panel", which consists of a plot plus one or more "strips" above it. The construction of such a panel is straightforward using grid, as the following code demonstrates (the output is shown in Figure 5:

```
grid.newpage()
lyt <-
  grid.layout(3, 1,
    heights=unit(c(1.5, 1.5, 1),
                 c("lines", "lines", "null")))
push.viewport(viewport(width=0.7,
```

```
                         height=0.7,
                         layout=lyt,
                         xscale=c(0.5, 8.5)))
push.viewport(viewport(layout.pos.row=1))
grid.rect(gp=gpar(fill="light green"))
grid.text("Strip 1")
pop.viewport()
push.viewport(viewport(layout.pos.row=2))
grid.rect(gp=gpar(fill="orange"))
grid.text("Strip 2")
pop.viewport()
push.viewport(viewport(layout.pos.row=3,
                       xscale=c(0.5, 8.5),
                       yscale=c(.1, .9)))
grid.rect()
grid.grill()
grid.points(unit(runif(5, 1, 8), "native"),
            unit(runif(5, .2, .8), "native"),
            gp=gpar(col="blue"))
grid.yaxis()
grid.yaxis(main=FALSE, label=FALSE)
pop.viewport()
grid.xaxis()
grid.xaxis(main=FALSE, label=FALSE)
```

This panel can now be included in a higher-level layout to produce an array of panels just as a scatterplot was placed within an array previously.

An important point about this example is that, once the command

```
push.viewport(viewport(layout.pos.row=1))
```

has been issued, drawing can occur within the context of the top strip, with absolutely no regard for any other coordinate systems in the graphic. For example, lines and rectangles can be drawn relative to an x-scale within this strip to indicate the value of a third conditioning variable and a text label can be drawn relative to the normalised coordinates within the strip — e.g., at location `x=unit(0, "npc")` with `just=c("left", "centre")` to left-align a label.

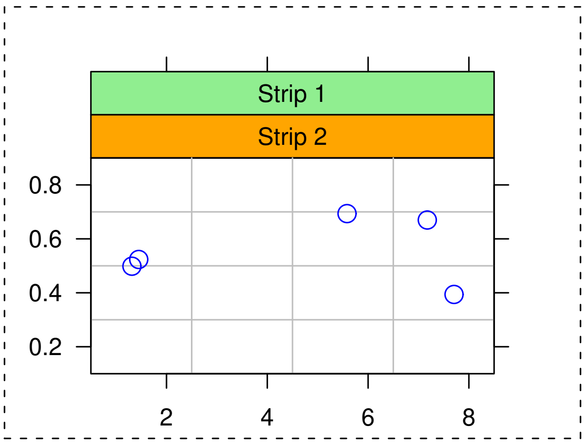


Figure 5: A Trellis-like panel produced using **grid**.

Given this sort of flexibility and power for combining graphical components, it becomes possible to seriously consider generating novel statistical graphics and producing custom images for the needs of individual data sets.

The following example, uses data from a study which was conducted to investigate the speed of cars at a location in Auckland, New Zealand. The variable of interest was the proportion of cars travelling above 60kph. This variable was recorded every hour for several days. In addition, the total volume of cars per hour was recorded. The researchers wanted to produce a graphic which presented the proportion data in a top panel and the volume data in a bottom panel as well as indicate day/night transitions using white versus grey bands and weekends or public holidays using white versus black strips.

The following code produces the desired graph, which is shown in Figure 6.

```
n <- dim(cardata)[1]
xrange <- c(0, n+1)
grid.newpage()
push.viewport(
  viewport(x=unit(3, "lines"),
           width=unit(1, "npc") -
                 unit(4, "lines"),
           y=unit(3, "lines"),
           height=unit(1, "npc") -
                  unit(5, "lines"),
           just=c("left", "bottom"),
           layout=grid.layout(5, 1,
                    heights=unit(rep(3, 1),
                      rep(c("mm", "null"),
                          length=5))),
           xscale=xrange, gp=gpar(fontsize=8)))
grid.rect(x=unit((1:n)[cardata$day == "night"],
                 "native"),
          width=unit(1, "native"),
          gp=gpar(col=NULL, fill="light grey"))
grid.rect()
grid.xaxis(at=seq(1, n, 24), label=FALSE)
grid.text(cardata$weekday[seq(1, n, 24)],
          x=unit(seq(1, n, 24)+12.5, "native"),
          y=unit(-1, "lines"))
draw.workday <- function(row) {
    push.viewport(viewport(layout.pos.row=row,
                           xscale=xrange))
    grid.rect(gp=gpar(fill="white"))
    x <- (1:n)[cardata$workday == "yes"]
    grid.rect(x=unit(x, "native"),
              width=unit(1, "native"),
              gp=gpar(fill="black"))
    pop.viewport()
}
draw.workday(1)
push.viewport(viewport(layout.pos.row=2,
                       xscale=xrange,
                       yscale=c(0.5, 1)))
grid.lines(unit(1:n, "native"),
           unit(cardata$prop, "native"))
grid.yaxis()
pop.viewport()
draw.workday(3)
push.viewport(
```

```
  viewport(layout.pos.row=4,
           xscale=xrange,
           yscale=c(0, max(cardata$total))))
grid.lines(unit(1:n, "native"),
           unit(cardata$total, "native"))
grid.yaxis()
pop.viewport()
draw.workday(5)
```

Some important points about this example are:

1. It is not impossible to do this using R's base graphics, but it is more "natural" using **grid**.

2. Having created this graphic using **grid**, arbitrary annotations are possible — all coordinate systems used in creating the unusual arrangement are available at the user-level for further drawing.

3. Having created this graphic using **grid**, it may be easily embedded in or combined with other graphical components.

Final remarks

The most obvious use of the functions in **grid** is in the development of new high-level statistical graphics functions, such as those in the **lattice** package. However, there is also an intention to lower the barrier for normal users to be able to build everyday graphics from scratch.

There are currently no complete high-level plotting functions in **grid**. The plan is to provide some default functions for high-level plots, but such functions inevitably have to make assumptions about what the user wants to be able to do — and these assumptions inevitably end up constraining what the user is able to achieve. The focus for **grid** will continue to be the provision of as much support as possible for producing complex statistical graphics by combining basic graphical components.

grid provides a number of features not discussed in this article. For information on those features and more examples of the use of **grid**, see the documentation at `http://www.stat.auckland.ac.nz/~paul/grid/grid.html`. **grid** was also described in a paper at the second international workshop on Distributed Statistical Computing (Murrell, 2001).

Bibliography

Bell lab's trellis page. URL `http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/`.

Richard A. Becker, William S. Cleveland, and Ming-Jen Shyu. The visual design and control of trellis display. *Journal of Computational and Graphical Statistics*, 5:123–155, 1996. 17

William S. Cleveland. *Visualizing data*. Hobart Press, 1993. ISBN 0963488406. 17

Paul Murrell. R Lattice graphics. In Kurt Hornik and Friedrich Leisch, editors, *Proceedings of the 2nd International Workshop on Distributed Statistical Computing, March 15-17, 2001, Technische Universität Wien, Vienna, Austria*, 2001. URL `http://www.ci.tuwien.ac.at/Conferences/DSC-2001/Proceedings/`. ISSN 1609-395X. 19

Murrell, Paul R. Layouts: A mechanism for arranging plots on a page. *Journal of Computational and Graphical Statistics*, 8:121–134, 1999. 16

Paul Murrell
University of Auckland, NZ
`paul@stat.auckland.ac.nz`

Lattice

An Implementation of Trellis Graphics in R

by Deepayan Sarkar

Introduction

The ideas that were later to become Trellis Graphics were first presented by Bill Cleveland in his book *Visualizing Data* (Hobart Press, 1993); to be later developed further and implemented as a suite of graphics functions in S/S-PLUS. In a broad sense, Trellis is a collection of ideas on how statistical graphics should be displayed. As such, it can be implemented on a variety of systems. Until recently, however, the only actual implementation was in S-PLUS, and the name Trellis is practically synonymous with this implementation.

lattice is another implementation of Trellis Graphics, built on top of R, and uses the very flexible capabilities for arranging graphical components provided by the **grid** add-on package. **grid** is discussed in a companion article in this issue.

In keeping with the R tradition, the API of the high-level Lattice functions are based on published descriptions and documentation of the S-PLUS Trellis Graphics suite. It would help to remember, however, that while every effort has been made to enable Trellis code in S-PLUS to run with minimal modification, Lattice is different from Trellis in S-PLUS in

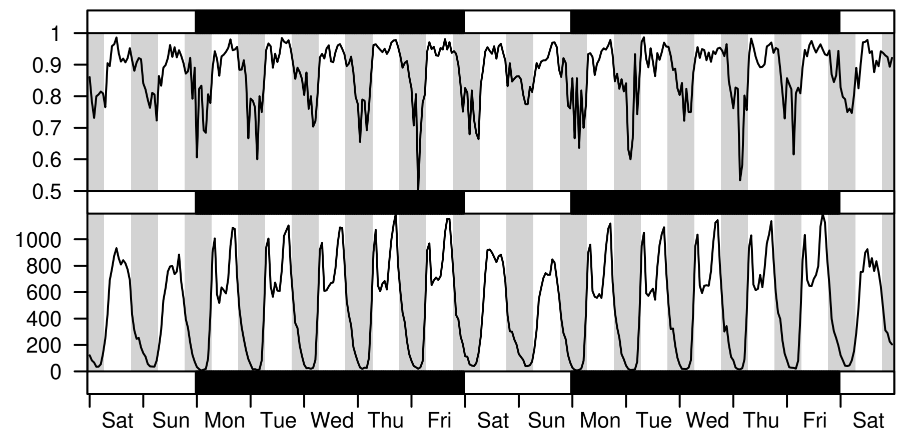


Figure 6: A custom plot produced using **grid**.

many respects; some unavoidable, some intentional.

The Trellis paradigm

Familiarity with some general principles Trellis uses in displaying statistical graphs can go a long way towards understanding it. The most important feature of Trellis Graphics is the ability to produce *multi-panel, multi-page* graphs — this was in fact the primary motivation (as well as the origin of the name) for Trellis. Perhaps no less important, however, is a *basic paradigm shift* from the way graphics is formulated and displayed conventionally in the S languages.

Conventional S graphics draws plots incrementally. For example, it is common to draw a histogram, then draw a box around it, add the axes, and perhaps add a few labels (`main`, `sub`, etc) to get the final plot. Trellis, on the other hand, would first create an object that would contain all the information necessary to create the plot (including what to draw, how to mark the axes, what `main`, `sub` are, if anything), and then plot it in one go.

This approach has certain advantages. For conventional S graphics, there is no way of knowing in advance whether the axes will be drawn, whether there will be a main label (and if so, how long or how large it would be), etc. Consequently, when the first plot is drawn, there has to be enough space left for everything else, in case the user decides to add these later. This is done by pre-allocating space for axis labels, main labels, etc. Obviously, the default amount of space is not always appropriate, and it is often necessary to change these spacings via the `par()` function. Trellis plots, however, take a different approach. A plot is drawn all at once, and thus exactly the requisite amount of space can be used for each component of the display.

Also unlike conventional S graphics, high level Lattice functions do not directly draw anything when they are called, but rather, they are more like other S functions in that they produce an object (of class `"trellis"`) which when `print`ed produces the actual plot. This enables changing several aspects of the plot with minimal recomputation via the `update.trellis` method, as well as the ability to save a plot for later plotting and draw the same plot in several devices. The `print` method for `"trellis"` objects also allows finer control over the placement of plots within the plotting region, in turn making possible more than one Trellis plot per page. This is similar to setting `par(mfrow)`, but slightly more flexible.

On the downside, this holistic approach to plotting can make calls to produce even slightly complicated plots look very intimidating at first (although the default display options are quite satisfactory for the most part, the need to change these also arises quite frequently).

Getting started

A few high level functions provide the basic interface to Lattice. Each of these, by default, produce a different kind of statistical graph. Scatter Plots are produced by `xyplot`, Box Plots and Dot Plots by `bwplot` and `dotplot`, Scatter Plot Matrices by `splom`, Histograms and Kernel Density Plots by `histogram` and `densityplot`, to name the most commonly used ones. These functions need, at a minimum, only one argument: a formula specifying the variables to be used in the plot, and optionally, the name of a data frame containing the variables in the formula.

The `formula` argument and conditioning

Unlike most conventional graphics functions, the variables in the plot are almost always specified in terms of a formula describing the structure of the plot. This is typically the unnamed first argument. For bivariate functions like `xyplot`, `bwplot`, etc, it is of the form `y~x | g1 * ··· * gn`. Here `x`, `y`, `g1`, ..., `gn` should be vectors of the same length, optionally in a data frame, in which case its name has to be given as the `data` argument.

The variables `g1`, ..., `gn` are optional, and are used to investigate the relationship of several variables through conditioning. When they are absent, the plots produced are similar to what can be done with conventional S graphics. When present, we get Conditioning Plots — separate 'panels' are produced for each unique combination of the levels of the conditioning variables, and only the subset of `x` and `y` that corresponds to this combination is used for displaying the contents of that panel. These conditioning variables are usually factors, but they can also be *shingles*, an innovative way of conditioning on continuous variables. Conditioning is illustrated in (the left part of) Figure 3, where dotplots of yields of different varieties of barley are displayed in different panels determined by the location of the experiment.

Conditioning can be done in all high level functions, but the form of the first part of the formula changes according to the type of plot produced. The typical form is `y ~ x`, where the `y` variable is plotted on the vertical axis, and `x` on the horizontal. For univariate functions like `histogram` and `densityplot`, where only the `x` variable is needed, the form of the first part is `~x`, and for trivariate functions like `cloud` and `levelplot` whose panels use three variables, the form is `z ~ x * y`.

Aspect ratio and banking

The information that the human eye can perceive from a plot can change significantly depending simply on the aspect ratio used for displaying the plot. Some Trellis functions can use 'banking' to automatically select a close to optimum aspect ratio that makes the plot more informative, when given the `aspect="xy"` argument. Figure 3 includes an example illustrating the usefulness of this feature.

Changing panel functions using grid

While the high level functions provide the ability to produce the most commonly used statistical graphs, it is often necessary to go beyond that and produce displays tailored for the problem at hand. Much of the power of Trellis comes from the ability to change the default display by means of a function supplied as the `panel` argument. This can be any function written using **grid**, and is called with the panel region set as the current viewport.

A word of caution: Conventional R graphics code will not work in the panel function. However, many predefined functions that can serve as building blocks of custom panel functions are already included in the Lattice package. (These usually suffice for *porting* Trellis code written for S-PLUS.)

An example below illustrates how to modify a call to `bwplot` to produce an *interaction plot*:

```
library(nlme)
data(Alfalfa)
levels(Alfalfa$Date) <-
    c('None', 'Sep 1', 'Sep 20', 'Oct 7')
bwplot(Yield ~ Date | Variety, Alfalfa,
       groups = Block, layout = c(3, 1),
       panel = "panel.superpose",
       panel.groups = "panel.linejoin",
       xlab = "Date of third cutting",
       main = "Block by Date interaction
               in 3 Varieties of Alfalfa")
```

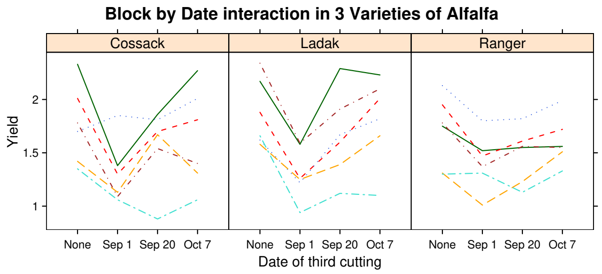


Figure 1: Interaction plot produced by `bwplot` with a custom panel function

A very useful feature of Lattice is that all arguments not recognized by high level functions are passed on to the panel function. This allows very general panel functions to be written, which can then be controlled by arguments given directly to the high level function.

Other commonly used features

All high level Lattice functions accept several arguments that modify various components of the display, some of which have been used in the examples given here. The most commonly used ones are briefly mentioned below, further details can be found in the documentation.

Panel Layout: When using conditioning variables, especially when a large number of panels are produced, it is important to arrange the panels in a nice and informative manner. While the default layout is usually satisfactory, finer control can be achieved via the `layout`, `skip`, `between` and `as.table` arguments.

Scales: An important component of the display is how the axis tick marks and labels are shown. This is controlled by the `scales` argument (`pscales` for `splom`).

Grouping variable: Apart from conditioning, another very common way to look at the relationship between more than two variables, particularly when one of them is a factor with a small number of levels, is by using different graphical parameters to distinguish between levels of this factor. This can be done using the `groups` argument, usually with `panel.superpose` as the panel function.

Handling multiple plots

The `print.trellis` function, when called explicitly with extra arguments, enables placing more than one Lattice plot on the same page. For example, the following code places two 3d scatter plots together to form a stereogram (figure 2):

```
data(iris)
print(cloud(Sepal.Length ~
            Petal.Length * Petal.Width,
            data = iris, perspective = FALSE,
            groups = Species,
            subpanel = panel.superpose,
            main = "Stereo",
            screen = list(z=20,x=-70,y=3)),
      split = c(1,1,2,1), more = TRUE)
print(cloud(Sepal.Length ~
            Petal.Length * Petal.Width,
            data = iris, perspective = FALSE,
            groups = Species,
            subpanel = panel.superpose,
            main = "Stereo",
            screen = list(z=20,x=-70,y=0)),
      split = c(2,1,2,1))
```

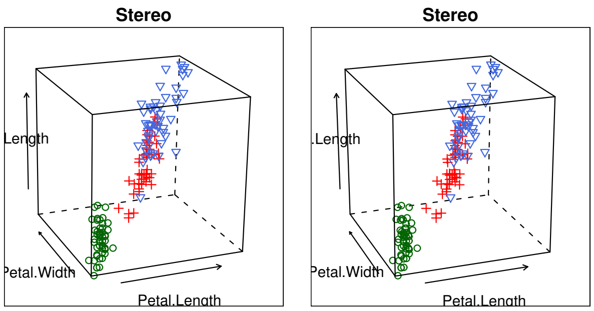


Figure 2: 3d Scatter plots (using `cloud`) of the Iris data, grouped by Species. Stare at the plot and focus your eyes behind the page so that the two images merge. The result should be the illusion of a 3D image.

Settings: Modifying the look and feel

The choice of various parameters, such as color, line type, text size, etc can change the look and feel of the plots obtained. These can be controlled to an extent by passing desired values as arguments named `col`, `lty`, `cex`, etc to high level functions. It is also possible to change the settings globally, by modifying a list that determines these parameters. The default settings are device specific — screen devices get a grey background with mostly light colors, while postscript output has a white background with darker colors. It is possible to change these settings, and to define entirely new 'themes' suited to particular tastes and purposes — for example, someone using the prosper package to create presentations in LaTeX might want to set all the foreground colors to white and use larger text. The functions `trellis.device`, `trellis.par.set`, `lset` and `show.settings` can be useful in this context. Lattice currently comes with one predefined theme that can be loaded by calling `lset(col.whitebg())`, and it is fairly easy to create new ones.

Using Lattice as a tool

Trellis was designed for a specific purpose, namely, to provide effective graphical presentation of the relationship between several variables through conditioning; and that is what it does best. It is a high level graphics package, and makes several assumptions about the form of the display it produces. In that respect, it is not always suitable as a tool for developing new graphical methods. It is quite flexibile in its own way, however, and can be particularly useful for creating custom display functions for specific types of data — the pkgnlme package, for example, uses Lattice functions extensively for its `plot` methods.

Conclusion

This article provides a brief overview of the functionality in the Lattice package, but is by no means a comprehensive introduction. For those interested in learning more, an excellent place to start is Bell Lab's Trellis webpage at `http://cm.bell-labs.com/cm/ms/departments/sia/project/trellis/`; although specific to the S-PLUS implementation, most of it is also applicable to Lattice. Gory details are of course available from the documentation accompanying the Lattice package, `?Lattice` is the recommended launching point. Finally, Lattice has its own webpage at `http://packages.r-project.org/lattice/`.

Deepayan Sarkar
University of Wisconsin, U.S.A.
`deepayan@stat.wisc.edu`

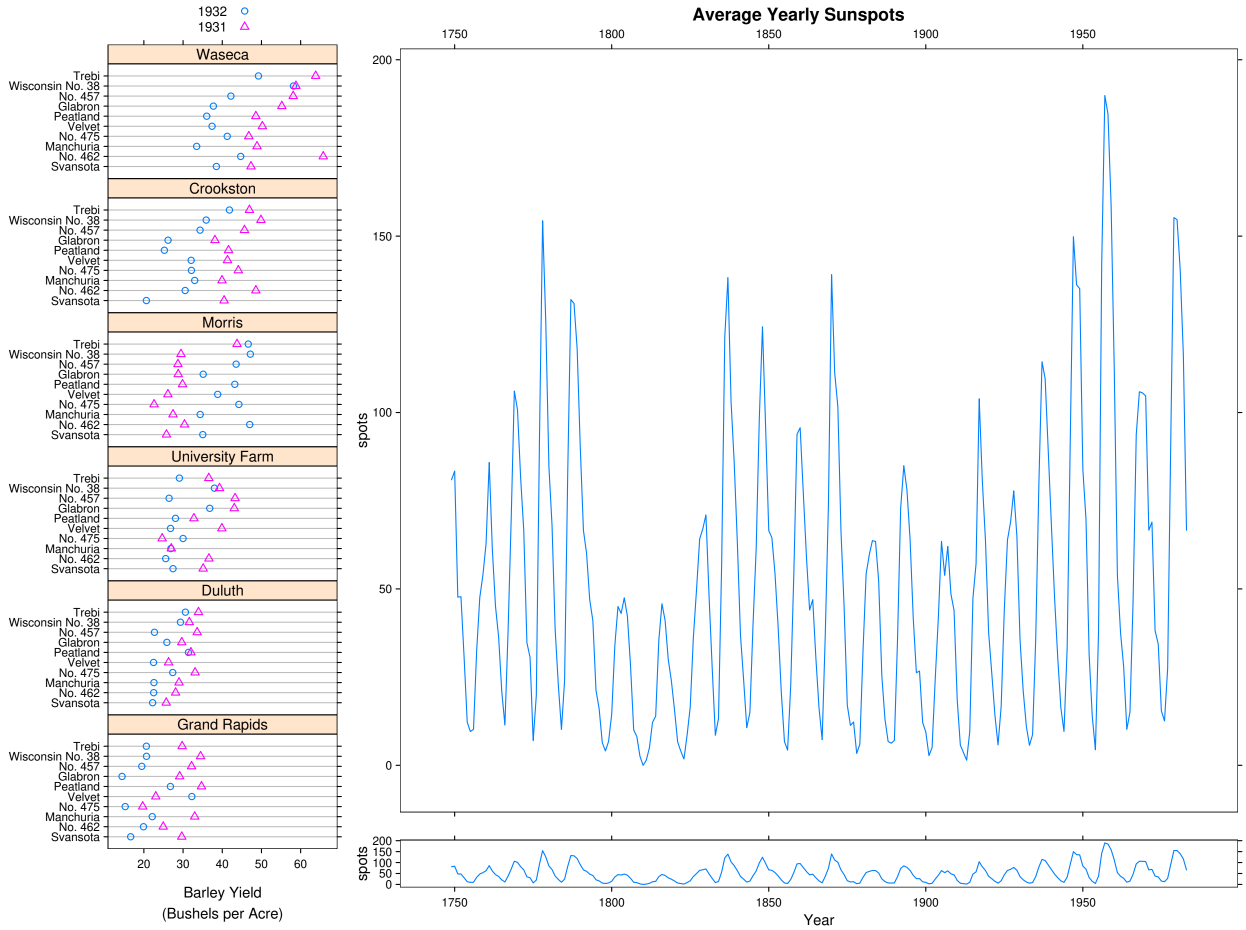


Figure 3: Two well known examples from Cleveland (1993). The left figure shows the yield of 10 different varieties of barley in 6 farms recorded for two consecutive years. The data has been around since Fisher, but this plot suggested a possible mistake in recording the data from Morris. The plots on the right show the usefulness of banking. The top figure is what an usual time series plot of the average yearly sunspots from 1749 to 1983 would look like. The bottom plot uses banking to adjust the aspect ratio. The resulting plot emphasizes an important feature, namely, that the ascending slope approaching a peak is steeper than the descending slope. [This plot was created using `print.trellis` for the layout. See accompanying code for details. The colour schemes used here, as well as in the other plots, are not the defaults. See the section on **Settings** for references on how to set 'themes'.]

```
data(barley)
plot1 <- dotplot(variety ~ yield | site, data = barley, groups = year, aspect = .5,
                 panel = "panel.superpose", panel.groups = "panel.dotplot", layout = c(1, 6),
                 col.line = c("grey", "transparent"), xlab = "Barley Yield\n(Bushels per Acre)",
                 key = list( text = list(c("1932", "1931")),
                 points = Rows(trellis.par.get("superpose.symbol"), 1:2)))
data(sunspots)
spots <- by(sunspots, gl(235, 12, lab = 1749:1983), mean)
plot2 <- xyplot(spots~1749:1983, xlab = "", type = "l", main = "Average Yearly Sunspots",
                scales = list(x = list(alternating = 2)),
plot3 <- xyplot(spots~1749:1983, xlab = "Year", type = "l", aspect = "xy")
print(plot1, position = c(0, 0, .3, ,1), more = TRUE)
print(plot2, position = c(.28, .12, 1, 1), more = TRUE)
print(plot3, position = c(.28, 0, 1, .13))
```

Programmer's Niche

Mind Your Language

Bill Venables

Introduction

John Chambers once told me that with S (including R) you have the choice of typing a command and implicitly presenting it to the evaluator or getting the engine to construct the command and explicitly presenting to the evaluator. The first is what everyone does when they use R and the second seemed to me at the time to be an esoteric possibility that might only be used in the most arcane situations imaginable. However it is not so. Language manipulation within R is a live possibility to consider for everyday data analysis and can be a boon to lazy typists. There is also one situation where without trying to do anything extreme or extraordinary you *must* use the second of John's alternatives to achieve the result you need.

In *S Programming*Venables and Ripley (2000) there is a discussion on language manipulation and its uses (see §3.4) which I do not want to repeat that here. Rather, what I would like to do is give a real example of a data analysis problem where a language manipulation solution offers an elegant, natural and practical solution. If you have the book you may wish to go back afterwards and read more, but the example should still make sense even if you have never heard of that book and intend never to go anywhere near it.

The problem

In biodiversity studies samples are taken from many different sites and, in the simplest case, a record is taken of whether or not each member of a suite of biological taxa is present or not. At the same time environmental variables are measured at each site to be used as predictor variables in building models for the probability that a taxon, say, is present at any site. In the particular case I will discuss here there are 1194 sites on the Great Barrier Reef from a study conducted by the Queensland Department of Primary Industry. The data frame containing the variables is called `QDPI`. There are 60 variables in the frame, the first three are location variables, the next 30 are possible environmental predictors and the final 27 are 0/1 variables indicating the presence/absence of 27 marine taxa.

We will consider separate models for each taxon. It would be natural to consider either logistic regression models or perhaps generalized additive models (in package **mgcv**) for example. For our purposes here tree models (using package **rpart**) have several advantages. So the problem is to construct separate tree models for each of 27 taxa where each model will have the full suite of 30 possible predictor variables from which to choose.

A strategy

Let's agree that we will put the fitted tree model objects in a list, say `tList`. We set up a dummy list to take them first

```
> nam <- names(QDPI)
> namX <- nam[4:33]
> namY <- nam[34:60]
> tList <- vector("list", length(namY))
> names(tList) <- namY
```

So far so good. Now for a bit of language manipulation. First we construct a character string version of an assignment to an element of `tList` of an `rpart` object, which will be evaluated at each cycle of a `for` loop:

```
> tAsgn <- paste("tList[[n]] <- try(rpart(X ~",
   paste(namX, collapse = " + "),
   ", QDPI, method = 'class'))", sep = "")
> tAsgn <- parse(text = tAsgn)[[1]]
```

(Notice how quotes within character strings can be handled by single quotes enclosed within doubles.) Turning character versions of commands into language objects is done by `parse(text =` *string*`)` but the result is an `expression` which in this case is rather like a list of language objects of length 1. It is not strictly necessary to extract the first object as we have done above, but it makes things look a little neater, at least. Here is the assignment object that we have constructed:

```
> tAsgn
tList[[n]] <- try(rpart(X ~ Depth +
     GBR.Bathyetry + GBR.Aspect +
     GBR.Slope + M.BenthicStress +
     SW.ChlorophyllA +
     SW.ChlorophyllA.SD + SW.K490 +
     SW.K490.SD + SW.BenthicIrradiance +
     OSI.CaCO3 + OSI.GrainSize +
     OSI.Rock + OSI.Gravel + OSI.Sand +
     OSI.Mud + CARS.Nitrate +
     CARS.Nitrate.SD + CARS.Oxygen +
     CARS.Oxygen.SD + CARS.Phosphate +
     CARS.Phosphate.SD + CARS.Salinity +
     CARS.Salinity.SD + CARS.Silica +
     CARS.Silica.SD + CARS.Temperature +
     CARS.Temperature.SD + Effort +
     TopographyCode, QDPI, method = "class"))
```

You can probably see now why I was not keen to type it all out. The index `n` will be supplied as a loop variable but the `X` dummy response variable will need to be replaced the by the *name* of each taxon in turn. Note that here *name* is a technical term. Language elements are composed of objects of a variety of special modes (and some, line numbers, not so special) and an object of mode "name" is the appropriate constituent of a language object to stand for a variable name.

Now for the main loop:

```
> for(n in namY) {
  TAsgn <- do.call("substitute",
  list(tAsgn, list(n = n, X = as.name(n))))
  eval(TAsgn)
  }
Error in matrix(c(rp$isplit[, 2:3], rp$dsplit),
  ncol = 5, dimnames = list(tname[rp$isplit[,:
  length of dimnames[1] not equal to array extent
```

There has been a problem with one of the modelling computations. We can find out which one it is by seeing which object does not have the correct mode at the finish:

```
> namY[which(sapply(tList, class) != "rpart")]
[1] "Anemone"
```

So Anemones were un able to have a tree model constructed for their presence/absence. [I am using an older version of R here and this seems to be a bug in pkgrpart that has since been corrected. It serves as an example here, though.]

Notes on `do.call`, `substitute` and `eval`

The three crucial functions we have used above, so far with no explanation are `do.call`, `substitute` and `eval`. We will not go into an extended explanation here but some pointers may be useful. First of all `do.call`. This used to be a well-kept secret but in recent years it has become fairly familiar to readers of R-news. It is perhaps the simplest to use function that constructs and evaluates a language object, so it is usually the first one people meet in their travels in R programming.

The function `do.call` is used to evaluate a function call where the name of the function is given as a character string as the first argument and the arguments to be used for the call to that function are given as a list in the second argument. It is probably useful to know that this second argument can be a list of objects, or a list of *names* of objects, or a mixture of both.

Nearly everyone knows of `substitute` through the idiom `deparse(substitute(arg))` for getting a character string version of the actual bit of code used for a formal argument, `arg`, on a call to a function. In this case `substitute(arg)` merely grabs the actual argument supplied without evaluating it and the `deparse(...)` step turns it back from a language object into a character string equivalent, essentially as it might have been typed.

This is only one use of `substitute` though. In general it may be used to take an expression and do a bit of surgery on it. Any of the *names* appearing in the first argument may be changed to something else, as defined by a second argument, which has to be a named list. Here is a much simpler example than the one above:

```
> substitute(a+b+c,
           list(a=1, c = as.name("foo")))
1 + b + foo
```

The small detail often overlooked in this kind of demonstration is that the first argument is *quoted*, that is, the surgery is done on the object verbatim. What happens if the thing we want to do surgery upon is a long, complicated expression that we are holding as a language object, such as the case above?

At first sight it may seem that the following should work:

```
> substitute(tAsgn, list(n = "Anemone",
                X = as.name("Anemone")))
tAsgn
```

but you can see the result is a disappointment. The quoting prevents the first argument from being evaluated. So how do we say "evaluate the first argument, dummy"? The solution is to use R to construct a call to `substitute` as we might have typed it and evaluate that. This is what `do.call` does, and here it is essential to use this indirect approach.

```
> do.call("substitute",
       list(tAsgn, list(n = "Anemone",
           X = as.name("Anemone"))))
tList[["Anemone"]] <- try(rpart(Anemone ~
    CARS.Nitrate + CARS.Nitrate.SD + ... +
    SW.K490 + SW.K490.SD + TopographyCode,
    QDPI, method = "class"))
```

The third in the trio of functions is `eval`. As the name suggests this can be used for explicitly submitting a language object to the evaluator for, well, evaluation within the current environment. There are more general and subtle facilities for modifying the environment using a second or third argument, but they need not bother us here. There is no deep mystery, but it is important to know that such a function exists.

Final comments

It is possible to think of simpler ways that might achieve the same result as ours. For example we could put all the `X` variables into one data frame, say `QDPI.X` and all the response variables into `QDPI.Y` and then use a loop like

```
> for(n in namY)
    tList[[n]] <- try(rpart(QDPI.Y[, n] ~ .,
            QDPI.X, method = "class"))
```

and it would work. The problem with this is that the objects we construct have a formula that has, literally `QDPI.Y[, n]` as the dependent variable in the formula. If we want to do anything with the objects afterwards, such as prune them, update them, &c, we need to re-establish what `n` is in this particular case. The original object `n` was the loop variable and that is long gone. This is not difficult, of course, but it is an extra detail we need to carry along that we don't need. Essentially the formula part of the object we generate would not be self-contained and this can cause problems.

The strategy we have adopted has kept all the variables together in one data frame and explicitly encoded the correct response variable by name into the formula of each object as we go. At the end each fitted `rpart` object may be manipulated in the usual way witout this complication involving the now defunct loop variable.

Bibliography

W. N. Venables and B. D. Ripley. *S Programming*. Springer-Verlag, New York, 2000. 24

Bill Venables
CSIRO Marine Labs, Cleveland, Qld, Australia
Bill.Venables@cmis.csiro.au

geoRglm: A Package for Generalised Linear Spatial Models

by Ole F. Christensen and Paulo J. Ribeiro Jr

geoRglm is a package for inference in generalised linear spatial models using Markov chain Monte Carlo (MCMC) methods. It has been developed at the Department of Mathematical Sciences, Aalborg University, Denmark and the Department of Mathematics and Statistics, Lancaster University, UK. A web site with further information can be found at http://www.maths.lancs.ac.uk/~christen/geoRglm. **geoRglm** is an extension to the **geoR** package (Ribeiro, Jr. and Diggle, 2001). Maximum compatibility between the two packages has been intended and **geoRglm** also uses several of **geoR**'s internal functions.

Generalised linear spatial models

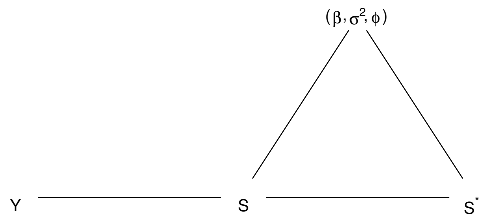
The classical geostatistical model assumes Gaussianity, which may be an unrealistic assumption for some data sets. The *generalised linear spatial model* (GLSM) as presented in Diggle et al. (1998), Zhang (2002) and Christensen and Waagepetersen (2002) provides a natural extension to deal with response variables for which a standard distribution other than the Gaussian more accurately describes the sampling mechanism involved.

The GLSM is a generalised linear mixed model in which the random effects are derived from a spatial process $S(\cdot)$. This leads to the following model specification.

Let $S(\cdot) = \{S(x) : x \in A\}$ be a Gaussian stochastic process with $\mathrm{E}[S(x)] = d(x)^{\mathrm{T}}\beta$, $\mathrm{Var}\{S(x)\} = \sigma^2$ and correlation function $\mathrm{Corr}\{S(x), S(x')\} = \rho(u;\phi)$ where $u = \|x - x'\|$ and ϕ is a parameter. Assume that the responses $Y_1, \ldots, Y_n$ observed at locations $x_1, \ldots, x_n$ in the sampling design, are conditionally independent given $S(\cdot)$, with conditional expectations $\mu_1, \ldots, \mu_n$, where $h(\mu_i) = S(x_i), i = 1, \ldots, n$, for a known link function $h(\cdot)$.

We write $S = (S(x_1), \ldots, S(x_n))^{\mathrm{T}}$ for the unobserved values of the underlying process at $x_1, \ldots, x_n$, and S^* for the values of $S(\cdot)$ at all other locations of interest, typically a fine grid of locations covering the study region.

The conditional independence structure of the GLSM is then indicated by the following graph.



The likelihood for a model of this kind is in general not expressible in closed form, but only as a high-dimensional integral

$$L(\beta, \sigma^2, \phi) = \int \prod_{i=1}^{n} f(y_i; h^{-1}(s_i)) p(s; \beta, \sigma^2, \phi) ds,$$

where $f(y;\mu)$ denotes the density of the error distribution parameterised by the mean μ, and $p(s;\beta,\sigma^2,\phi)$ is the multivariate Gaussian density for the vector S. The integral above is also the normalising constant in the conditional distribution $[S|y, \beta, \sigma^2, \phi]$,

$$p(s \mid y, \beta, \sigma^2, \phi) \propto \prod_{i=1}^{n} f(y_i; h^{-1}(s_i))\, p(s; \beta, \sigma^2, \phi).$$

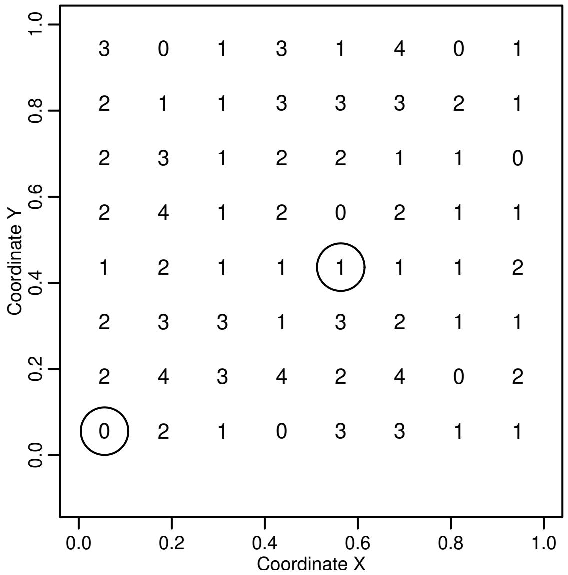
In practice, the high dimensionality of the integral prevents direct calculation of the predictive distribution $[S^* \mid y, \beta, \sigma^2, \phi]$. Markov chain Monte Carlo provides a solution to this. First by simulating a Markov chain we obtain a sample $s(1), \ldots, s(m)$ from $[S|y, \beta, \sigma^2, \phi]$, where each $s(j)$ is an n-dimensional vector. Second, by direct sampling from $[S^*|s(j), \beta, \sigma^2, \phi]$, $j = 1, \ldots, m$ we obtain a sample $s^*(1), \ldots, s^*(m)$ from $[S^*|y, \beta, \sigma^2, \phi]$. The MCMC algorithm uses Langevin-Hastings updates of S which are simultaneous updates based on gradient information.

In a Bayesian analysis priors must be assigned to the parameters in the model. For (β, σ^2) a conjugate prior exists such that these parameters can be integrated out analytically, whereas for ϕ one has to extend the MCMC algorithm above with updates of this parameter. We use a Metropolis random walk-type proposal for updating ϕ.

In its current version **geoRglm** implements the spatial Poisson model and the spatial binomial model.

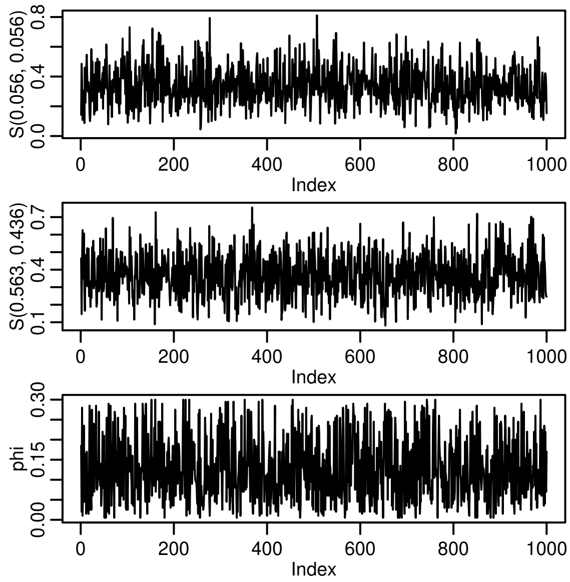
Package features

The following example gives a short demonstration of an analysis for a binomial spatial model with logistic link using the function `binom.krige.bayes`. We omit the specific commands here, but refer to the **geoRglm** homepage for further details. Consider the simulated data set shown below which consists of binomial data of size 4 at 64 locations.

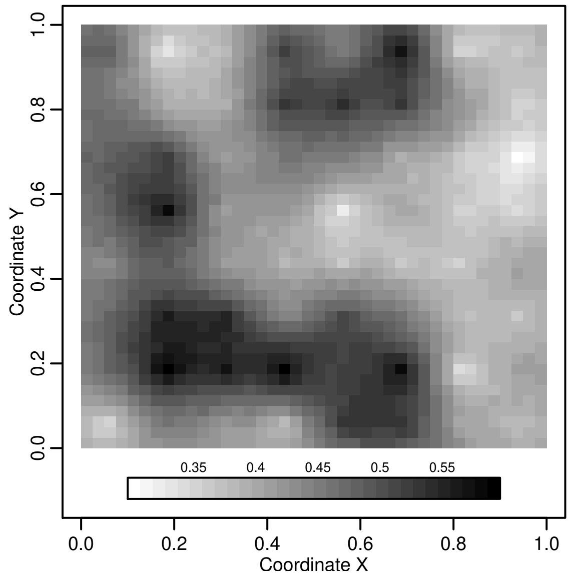


Priors for the parameters and options for the MCMC algorithm are set using the `prior.glm.control` and `mcmc.control` functions, respectively. As a rule of thumb the proposal variances must be tuned such that the acceptance rates for updating the random effects and the parameter ϕ are approximately 60% and 25%, respectively.

Output from the MCMC algorithm is presented below for the parameter ϕ and for the two random effects at locations marked with a circle in the figure above.



Predicted values of the probabilities $p(x) = \exp(S(x))/(1 + \exp(S(x)))$ at 1600 locations are plotted below using the function `image.kriging` from **geoR**.



Further details about this example and an introduction to the models can be found in Diggle et al. (2002) and in the files 'inst/doc/bookchap.pdf' and 'inst/doc/geoRglm.intro.pdf' distributed with the package.

Future developments

Work in progress with Gareth Roberts and Martin Sköld aims to improve convergence and mixing of the MCMC algorithm by using a more appropriate parameterisation.

Acknowledgements

Some of the C code in the package is based on code originally developed together with Rasmus Waagepetersen. We are grateful to Peter J. Diggle for giving encouragement and support to the development of the package. Ole acknowledges support from DINA, NERC and the EU TMR network. Paulo acknowledges CAPES/Brasil grant 1676/96-2.

Bibliography

O.F. Christensen and R.P. Waagepetersen. Bayesian prediction of spatial count data using generalized linear mixed models. *Biometrics*, 58:280–286, 2002. 26

P. J. Diggle, P. J. Ribeiro Jr, and O. F. Christensen. An introduction to model-based geostatistics. In M. B. Hansen and J. Møller, editors, *Spatial statistics and computational methods*. Springer Verlag, 2002. (to appear). 27

P. J. Diggle, J. A. Tawn, and R. A. Moyeed. Model-based geostatistics (with discussion). *Appl. Statist.*, 47:299–350, 1998. 26

Paulo J. Ribeiro, Jr. and Peter J. Diggle. geoR: A package for geostatistical analysis. *R News*, 1(2):14–18, 2001. 26

H. Zhang. On estimation and prediction for spatial generalised linear mixed models. *Biometrics*, 58:129–136, 2002. 26

Ole F. Christensen
Lancaster University, UK
o.christensen@lancaster.ac.uk

Paulo J. Ribeiro Jr
Universidade Federal do Paraná, Brasil
and *Lancaster University, UK*
Paulo.Ribeiro@est.ufpr.br

Querying PubMed

Web Services

by Robert Gentleman and Jeff Gentry

Introduction

While many view the world wide web (WWW) as an interactive environment primarily designed for interactive use, more and more sites are providing web services that can be accessed programmatically. In this article we describe some preliminary tools that have been added to the *annotate* package in the Bioconductor project www.bioconductor.org. These tools facilitate interaction with resources provided at the National Center for Biotechnology Information (NCBI) located at www.ncbi.nlm.nih.gov. These ideas represent only a very early exploration of a single site and we welcome any contributions to the project in the form of enhancements, new tools, or tools adapted to other sites providing web services.

We believe that web services will play a very important role in computational biology. In part this is because the data are complex and gain much of their relevance by association with other data sources. For example, knowing that there is a particularly high level of messenger RNA (mRNA) for some gene (or set of genes) does not provide us with much insight. However, associating these genes with the relevant scientific literature and finding common themes often does provide new insight into how these genes interact.

We can think of a cellular pathway as a set of genes that interact (through the proteins that they produce) to provide a particular function or protein. A second way of obtaining insight into the role of certain genes would be to examine the expression of mRNA for a set of genes in a particular pathway, or to take a set of genes and determine whether there is a particular pathway that contains (most of) these genes.

Both of these examples rely on associating experimental data with data that are available in databases or in textual form. These latter data sources are often large and are continually evolving. Thus, it does not seem practical nor prudent to keep local versions of them suitable for querying. Rather, we should rely on retrieving the data when it is wanted and on tools to process the data that are obtained from on-line sources.

It is important to note that most of the processes we are interested in can be carried out interactively. However, there are two main advantages to designing programmatic interfaces. First, interactive use introduces a rate limiting step. The analysis of genomic data needs to be high throughput. A second reason to prefer programmatic access is that it allows for the possibility of combining data from several sources, possibly filtered through online resources, to provide a new product. Another reason to prefer a programmatic approach is that it makes fewer mistakes and gets bored or distracted somewhat less easily than a human.

In this article we describe the features added to the **annotate** package that facilitate interactions with abstracts made available from the PubMed archive supplied by the National Library of Medicine. There are many other functions in **annotate** such as `genbank` and `locuslinkQuery` that provide web services tools that we will not discuss here.

Querying PubMed

We have written very little code to provide a very powerful tool for analysing textual information but rather have relied on some well written tools to provide basic services. Our tools are still in development and are likely to change over the next few months as we gain experience but we believe that the ideas are of interest to many researchers in the area of Bioinformatics and computational biology.

We will focus on the analysis of microarray data but this is just to provide some specific examples, the ideas are much more widely applicable. A very simplified view of the biology is that DNA makes RNA which in turn is translated into proteins. Proteins and their interactions basically provide mechanisms for all cellular functions and hence are of great interest.

DNA microarray experiments give us a static view of the level of messenger RNA in a set of biological samples. We are often interested in finding sets of mRNA that are highly expressed or not expressed in particular subsets of our samples. A typical experiment involves the study of a small number of samples (usually less than 100) and a large number of genes (usually more than 10,000).

There are several reasons why we would like to bring more biologically relevant information into the data analytic process. This can be done in many ways, one of which we will discuss next. There is a large repository of biological knowledge available in the published scientific literature. We are becoming more capable of processing and dealing with that literature in some specific ways.

We have exploited connections, a publicly available resource PubMed and the **XML** package to construct a tool for obtaining PubMed abstracts in a standard format. This is possible since PubMed provides tools for downloading their data in XML format. This makes the processing on our end much simpler and less error prone.

Once we have the abstracts we can process them in many ways. We can provide the data analyst with a set of titles or of keywords or of any other components. In addition, if the abstract is provided we can do some plain text searching of it using tools such as `grep` and `regexp`. In some cases the text of the entire document is available and that also can be obtained electronically and processed in a similar fashion.

In the **annotate** package we have produced a simple interface from microarray data to the PubMed abstracts. A user can select a set of genes that they find interesting and map these to an appropriate set of identifiers. These identifiers are then used to obtain the PubMed identifiers for associated abstracts. Finally PubMed is queried and these abstracts are downloaded into a particular form than can be processed using standard tools in R.

The code for a simple example is given below. In this example we have simply selected some *interesting genes* and then retrieved the abstracts and searched them for the word *protein*.

```
library(Biobase)
library(annotate)
## load up our test data set
data(eset)
## generally these would come from a filtering
## or other selection procedure
int.genes <- geneNames(eset)[273:283]
absts <- pm.getabst(int.genes, "hgu95A")
pm.titles(absts)
## which abstracts mention the word protein
wh.Protein <- sapply(absts,
   function(x) pm.abstGrep("[Pp]rotein", x))
```

It would also be nice to go in the other direction and we are actively developing tools that will help a researcher go from a gene of interest to a set of related genes. These can then be used to examine the available data to see if it concurs with the scientific literature.

This is a work in progress in almost every sense. Our programs need to be further developed. PubMed is growing and expanding the data resources it provides. New text processing algorithms are being developed that we would like to adapt to R so that they can be used to provide context for some of the queries being made. We believe that tools of this nature will become essential for computational biology.

The amount of processing that can currently be done is limited by a lack of algorithms and available data. There are a large number of data bases that exist but for which there is no programmatic way to access the data they contain or to perform queries remotely. Essentially we hope that more database providers see themselves as web services providers rather than as interactive tool providers. Making a tool interactive introduces a rate limiting step in our search for high throughput analysis tools. On the algorithm front it will be essential to start exploring ways of providing contextual information. We are currently limited to testing whether a gene's name (in one of its many forms) appears but cannot easily tell whether it is a passing reference or if there is substantial information being provided.

The details

The ability of the query functions to interact with the NCBI databases is provided by a set of well documented utilities provided by NCBI that work through standard HTTP connections. We describe some of them here, but refer the interested reader to the NCBI web site for definitive documentation.

All of our query functions can have the results of the query returned as an R object or have it rendered in the user's web browser. The NCBI resources are queried through CGI scripts which take various '&' separated parameters.

The LocusLink functions locuslinkByID and locuslinkQuery provide the ability to perform LocusLink searches either via a set of specific ID values or an actual text search respectively. In the former case, the URL to use is simple, http://www.ncbi.nih.gov/LocusLink/LocRpt.cgi?l=<id1>,<id2>,<id3>.... For all comma separated lists we use the HTML symbol '%2c' to circumvent browser issues.

The ability to run a text search is slightly more complicated, as the LocusLink database is divided by species. The R function locuslinkQuery takes a text query, and a set of zero or more species (the default is HS, human). Each supplied species is pasted to the string '&ORG=', and then appended to the query itself. Finally this string is pasted to the base URL to yield: http://www.ncbi.nih.gov/LocusLink/list.cgi?Q=<query>&ORG=<S1>&ORG=<S2>.... This URL is then sent to the user's browser, and the proper LocusLink page is displayed. The interactions with LocusLink are limited because LocusLink does not provide for programmatic querying.

The pubmed and genbank functions both utilize a set of CGI tools known as *Entrez*. *Entrez* provides the same basic searching mechanisms that are available to the interactive user, as well as a set of utilities designed for downloading the results programmatically.

To render the output into the user's browser we use the query tool provided by *Entrez*. query understands either Pubmed ID values or Genbank accession numbers, although the structure of the query is slightly different depending on which is used. For Pubmed the option is,

```
cmd=Retrieve&db=<id1>,<id2>...
```

and for Genbank it is

```
cmd=Search&db=<acc1>,<acc2>...
```

In either case, this is attached to form the full URL http://www.ncbi.nih.gov/entrez/query.fcgi?tool=bioconductor&cmd=Retrieve&db=<id1>.... Note the use of the tool directive, which is requested by NCBI to mark automated usage of their *Entrez* tools to help them track usage information, etc.

Other forms of output are available if requests are made using the pmfetch tool provided by *Entrez*. Since XML is one of the formats available and R has the **XML** package (D. Temple Lang) we prefer to obtain the data in XML. The results of a query are a list of XML objects that can be further processed. The interactive tools discussed above (pm.getAbst) rely on pubmed. The pmFetch utility has four different options that can be set by the user. The first is what format to display the data in, and is noted in the URL by report=*type*, where *type* is one of a variety of options (e.g., brief, Medline, Docsum, xml). The next option determines what format the data should be rendered in (text, file, or html — which is the default) and is set using mode=*type*. Third is a field to set which NCBI database to retrieve the data from (PubMed, Protein, Nucleotide, or Popset), and is called with db=*type*. Lastly, one needs to specify which IDs to use, which can be either PubMed IDs (PMID), MEDLINE Identifiers (UI), or molecular biology database (GI) and this is done with the command id=<id1>,<id2>,... or id=<id1>&<id2>&.... Note that these four parameters can be in any order within the URL and are separated using the '&' symbol.

For these functions we are always using text mode and a display value of XML. The database flag is currently set to either PubMed or Nucleotide depending on if this is being constructed by pubmed or genbank and we are using the comma separated method for the ID flags. As an example, if one were to make the call within R:

```
pubmed("11780146","11886385","11884611",
       disp="data")
```

the actual URL constructed to be sent to NCBI will be:

```
http://www.ncbi.nih.gov/entrez/utils/pmfetch.
fcgi?report=xml&mode=text&tool=bioconductor&db=
PubMed&id=11780146\%2c11886385\%2c11884611
```

Opening an http connection to this URL would provide the requested data. This is stored in XML objects and a list returned to the user for further processing.

Acknowledgment

Robert Gentleman's work is supported in part by NIH/NCI Grant 2P30 CA06516-38.

Robert Gentleman
DFCI
rgentlem@jimmy.harvard.edu

Jeff Gentry
DFCI
jgentry@jimmy.harvard.edu

evd: Extreme Value Distributions

by Alec Stephenson

Extreme value distributions arise as the limiting distributions of normalized maxima. They are often used to model extreme behaviour; for example, joint flooding at various coastal locations.

evd contains simulation, distribution, quantile and density functions for univariate and multivariate parametric extreme value distributions.

It also provides functions that calculate maximum likelihood estimates for univariate and bivariate models.

A user's guide is included in the package. It can also be downloaded directly (in postscript or pdf) from `http://www.maths.lancs.ac.uk/~stephena/`.

Introduction

Let $X_1, \ldots, X_m$ be *iid* random variables with distribution function F. Let $M_m = \max\{X_1, \ldots, X_m\}$. Suppose there exists normalizing sequences a_m and b_m such that $a_m > 0$ and as $m \to \infty$

$$\Pr(Z_m \leq z) = [F(a_m z + b_m)]^m \to G(z)$$

for $z \in \mathbb{R}$, where G is a non-degenerate distribution function and $Z_m = (M_m - b_m)/a_m$ is a sequence of normalized maxima. It follows that the distribution function G is generalized extreme value, namely

$$G(z) = \exp\left[-\left\{1 + \xi\left(z - \mu\right)/\sigma\right\}_+^{-1/\xi}\right],$$

where (μ, σ, ξ) are location, scale and shape parameters, $\sigma > 0$ and $h_+ = \max(h, 0)$. The case $\xi = 0$ is defined by continuity.

Multivariate extreme value distributions arise in a similar fashion (see Kotz and Nadarajah (2000) for details). In particular, any bivariate extreme value distribution can be expressed as

$$G(z_1, z_2) = \exp\left\{-(y_1 + y_2) A\left(\frac{y_1}{y_1 + y_2}\right)\right\},$$

where

$$y_j = y_j(z_j) = \{1 + \xi_j(z_j - \mu_j)/\sigma_j\}_+^{-1/\xi_j}$$

for $j = 1, 2$. The dependence function A characterizes the dependence structure of G. $A(\cdot)$ is a convex function on $[0, 1]$ with $A(0) = A(1) = 1$ and $\max(\omega, 1 - \omega) \leq A(\omega) \leq 1$ for all $0 \leq \omega \leq 1$. The jth univariate marginal distribution is generalized extreme value, with parameters (μ_j, σ_j, ξ_j).

Parametric models for the dependence function are commonly used for inference. The logistic model appears to be the most widely used. The corresponding distribution function is

$$G(z_1, z_2; \alpha) = \exp\left\{-(y_1^{1/\alpha} + y_2^{1/\alpha})^\alpha\right\}, \qquad (1)$$

where the dependence parameter $\alpha \in (0, 1]$. Independence is obtained when $\alpha = 1$. Complete dependence is obtained as $\alpha \downarrow 0$. Non-parametric estimators of A also exist, most of which are based on the estimator of Pickands (1981).

Features

- Simulation, distribution, quantile, density and fitting functions for the generalized extreme value and related models. This includes models such as $[F(\cdot)]^m$ for a given integer m and distribution function F, which enable e.g. simulation of block maxima.

- Simulation, distribution, density and fitting functions for eight parametric bivariate extreme value models. Non-parametric estimates of the dependence function can also be calculated and plotted.

- Simulation and distribution functions for two parametric multivariate extreme value models.

- Linear models for the generalized extreme value location parameter(s) can be implemented within maximum likelihood estimation. (This incorporates the forms of non-stationary most often used in the literature.)

- All fitting functions allow any of the parameters to be held fixed, so that nested models can easily be compared.

- Model diagnostics and profile deviances can be calculated/plotted using `plot`, `anova`, `profile` and `profile2d`.

Application

The `sealevel` data frame (Coles and Tawn, 1990) is included in the package. It has two columns containing annual sea level maxima from 1912 to 1992 at Dover and Harwich, two sites on the coast of Britain. There are 39 missing maxima in total; nine at Dover and thirty at Harwich.

The maxima on both margins appear to be increasing with time. The following snippet fits the logistic model (1) with simple linear trend terms on each marginal location parameter.

```
data(sealevel) ; sl <- sealevel
tt <- (1912:1992 - 1950)/100
lg <- fbvlog(sl, nsloc1 = tt, nsloc2 = tt)}
```

The fitted model contains maximum likelihood estimates for $(\gamma_1, \beta_1, \sigma_1, \xi_1, \gamma_2, \beta_2, \sigma_2, \xi_2, \alpha)$ where, for the ith observation, the marginal location parameters are

$$\mu_j(i) = \gamma_j + \beta_j t_i$$

for $j = 1, 2$ and $t_i =$ `tt[i]`. The significance of the trend parameters can be tested using the following analysis of deviance.

```
lg2 <- fbvlog(sl)
anova(lg, lg2)
```

This yields a p-value of about 10^{-6} for the hypothesis $\beta_1 = \beta_2 = 0$.

More complex models for the marginal location parameters can be fitted. Further tests suggest that a quadratic trend could be implemented for the Harwich maxima, but we retain the model `lg` for further analysis. The profile deviance of the dependence parameter α from (1), which corresponds to element `"dep"` in `lg$estimate`, can be produced using `profile`, as shown below.

```
pr <- profile(lg, "dep", xmax = 1)
plot(pr)
```

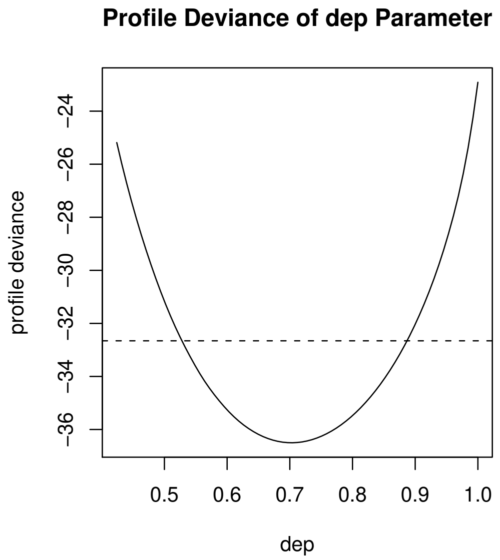


Figure 1: Profile deviance for α.

The horizontal line on the plot represents the 95% confidence interval $(0.53, 0.89)$, which can be calculated explicitly by `pcint(pr)`.

Diagnostic plots for the dependence structure and for the generalized extreme value margins can be produced as follows. The diagnostic plots for the dependence structure include Figure 2, which compares the fitted estimate of the dependence function A to the non-parametric estimator of Capéraà et al. (1997).

```
plot(lg)
plot(lg, mar = 1)
plot(lg, mar = 2)
```

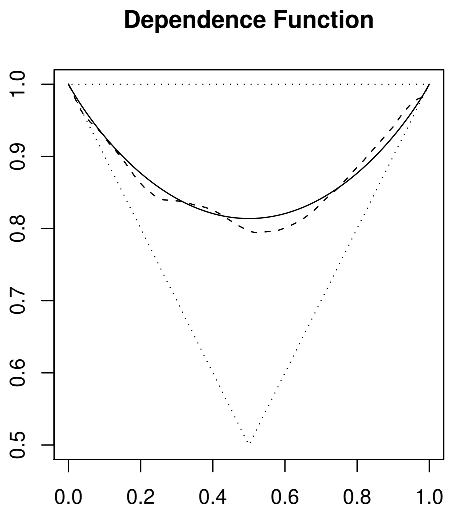


Figure 2: Estimates for the dependence function $A(\cdot)$; the logistic model (solid line), and the non-parametric estimator of Capéraà et al. (1997) (dashed line). The dotted border represents the constraint $\max(\omega, 1-\omega) \le A(\omega) \le 1$ for all $0 \le \omega \le 1$.

Alternative parametric models for the dependence structure can be fitted in a similar manner. The logistic model has the lowest deviance (evaluated at the maximum likelihood estimates) amongst those models included in the package that contain one dependence parameter. The models that contain two dependence parameters produce similar fits compared to the logistic. The models that contain three dependence parameters produce unrealistic fits, as they contain a near singular component. Amongst all models included in the package, the logistic is seen to give the best fit under a number of widely used criteria.

Bibliography

P. Capéraà, A.-L. Fougères, and C. Genest. A non-parametric estimation procedure for bivariate extreme value copulas. *Biometrika*, 84:567–577, 1997. 32

S. G. Coles and J. A. Tawn. Statistics of coastal flood prevention. *Phil. Trans. R. Soc. Lond., A*, 332:457–476, 1990. 31

S. Kotz and S. Nadarajah. *Extreme Value Distributions*. Imperial College Press, London, 2000. 31

J. Pickands. Multivariate extreme value distributions. *Proc. 43rd Sess. Int. Statist. Inst.*, 49:859–878, 1981. 31

Alec Stephenson
Lancaster University, UK
`a.stephenson@lancaster.ac.uk`

ipred: Improved Predictors

by Andrea Peters, Torsten Hothorn and Berthold Lausen

Introduction

In classification problems, there are several attempts to create rules which assign future observations to certain classes. Common methods are for example linear discriminant analysis or classification trees. Recent developments lead to substantial reduction of misclassification error in many applications. Bootstrap aggregation ("bagging", Breiman, 1996a) combines classifiers trained on bootstrap samples of the original data. Another approach is indirect classification, which incorporates a priori knowledge into a classification rule (Hand et al., 2001). Since the misclassification error is a criterion to assess the classification techniques, its estimation is of main importance. A nearly unbiased but highly variable estimator can be calculated by cross validation. Efron and Tibshirani (1997) discuss bootstrap estimates of misclassification error. As a by-product of bagging, Breiman (1996b) proposes the out-of-bag estimator.

However, the calculation of the desired classification models and their misclassification errors is often aggravated by different and specialized interfaces of the various procedures. We propose the **ipred** package as a first attempt to create a unified interface for improved predictors and various error rate estimators. In the following we demonstrate the functionality of the package in the example of glaucoma classification. We start with an overview about the disease and data and review the implemented classification and estimation methods in context with their application to glaucoma diagnosis.

Glaucoma

Glaucoma is a slowly processing and irreversible disease that affects the optic nerve head. It is the second most reason for blindness worldwide. Glaucoma is usually diagnosed based on a reduced visual field, assessed by a medical examination of perimetry and a smaller number of intact nerve fibers at the optic nerve head.

One opportunity to examine the amount of intact nerve fibers is using the Heidelberg Retina Tomograph (HRT), a confocal laser scanning tomograph, which does a three dimensional topographical analysis of the optic nerve head morphology. It produces a series of 32 images, each of 256×256 pixels, which are converted to a single topographic image, see Figure 1. A less complex, but although a less informative examination tool is the 2-dimensional fundus photography.

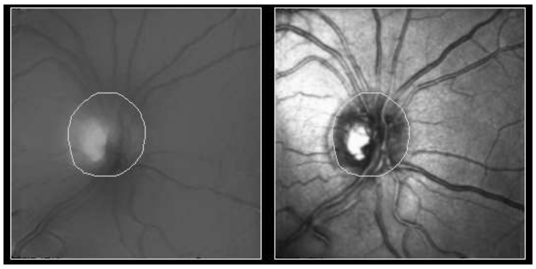


Figure 1: Topographic image of the optic nerve head, obtained by an examination with the Heidelberg Retina Tomograph.

However, in cooperation with clinicians and a priori analysis we derived a diagnosis of glaucoma based on three variables only: w_{lora} represents the loss of nerve fibers and is obtained by a 2-dimensional fundus photography, w_{cs} and w_{clv} describe the visual field defect (Peters et al., 2002).

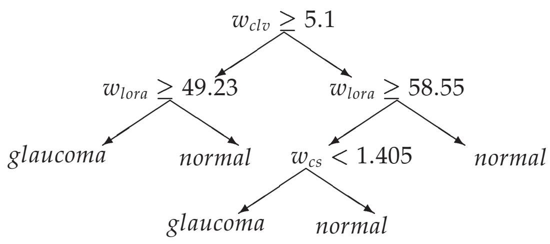


Figure 2: Glaucoma diagnosis.

Figure 2 represents the diagnosis of glaucoma in terms of a medical decision tree. A complication of the disease is that a damage in the optic nerve head morphology precedes a measurable visual field defect. Furthermore, an early detection is of main importance, since an adequate therapy can only slow down the progression of the disease. Hence, a classification rule for detecting early damages should include morphological informations, rather than visual field data only. Therfore, we construct classification rules based on 64 HRT variables and 7 anamnestic variables to predict the class membership of future observations. We use data accumulated at the Erlanger Eye Hospital (Hothorn et al., 2002) and match 170 observations of 85 normal and 85 glaucoma eyes by age and sex to prevent for possible confounding.

Bootstrap aggregation

Referring to the example of glaucoma diagnosis we first demonstrate the functionality of `bagging` and `predict.bagging`. We fit a bootstrap aggregated classification tree with `nbagg = 50` bootstrap replications by

```
R> fit <- bagging(diagnosis ~ ., nbagg = 50,
+   data = study.group, coob=TRUE)
```

where `study.group` contains explanatory HRT and anamnestic variables and the response of glaucoma diagnosis, a factor at two levels `normal` and `glaucoma`. `print.bagging` returns informations about the `bagging` object, i.e., the number of bootstrap replications used and, as requested by `coob=TRUE`, the out-of-bag estimate of misclassification error (Breiman, 1996b).

```
R> fit
Bagging classification trees
with 50 bootstrap replications

Out-of-bag misclassification error: 0.2242
```

The out-of-bag estimate uses the observations which are left out in a bootstrap sample to estimate the misclassification error at almost no additional computational costs. Hothorn and Lausen (2002) propose to use the out-of-bag samples for a combination of linear discriminant analysis and classification trees, called "Double-Bagging", which is available by choosing `method="double"`.

`predict.bagging` predicts future observations according to the fitted model.

```
R> predict(fit,
+    newdata=study.group[c(1:3, 86:88), ])
[1] normal   normal   normal
                 glaucoma glaucoma glaucoma
Levels:  glaucoma normal
```

Both `bagging` and `predict.bagging` rely on the `rpart` routines. The `rpart` routine for each bootstrap sample can be controlled in the usual way. By default `rpart.control` is used with `minsplit=2` and `cp=0`. The function `prune.bagging` can be used to prune each of the trees in a `bagging` object to an appropriate size.

Indirect classification

Especially in a medical context it often occurs that a priori knowledge about a classifying structure is given. For example it might be known that a disease is assessed on a subgroup of the given variables or, moreover, that class memberships are assigned by a deterministically known classifying function. Hand et al. (2001) proposes the framework of indirect classification which incorporates this a priori knowledge into a classification rule. In this framework we subdivide a given data set into three groups of variables: those to be used predicting the class membership (explanatory), those to be used defining the class membership (intermediate) and the class membership variable itself (response). For future observations, an indirect classifier predicts values for the appointed intermediate variables based on explanatory variables only. The observation is classified based on their predicted intermediate variables and a fixed classifying function. This indirect way of classification using the predicted intermediate variables offers possibilities to incorporate a priori knowledge by the subdivision of variables and by the construction of a fixed classifying function.

We apply indirect classification by using the function `inclass`. Referring to the glaucoma example, explanatory variables are HRT and anamnestic variables only, intermediate variables are w_{lora}, w_{cs} and w_{clv}. The response is the diagnosis of glaucoma which is determined by a fixed classifying function and therefore not included in the learning sample `study.groupI`. We assign the given variables to explanatory and intermediate by specifying the input formula.

```
R> formula.indirect <- clv + lora + cs ~ .
```

The variables on the left-hand side represent the intermediate variables, modeled by the explanatory variables on the right-hand side. Almost each modeling technique can be used to predict the intermediate variables. We chose a linear model by `pFUN = lm`.

```
R> fit <- inclass(formula.indirect,
+    pFUN = lm, data = study.groupI)
```

`print.inclass` displays the subdivision of variables and the chosen modeling technique

```
R> fit
Indirect classification, with 3
             intermediate variables:
clv lora cs

Predictive model per intermediate is lm
```

Indirect classification predicts the intermediate variables based on the explanatory variables and classifies them according to a fixed classifying function in a second step, that means a deterministically known function for the class membership has to be specified. In our example this function is given in Figure 2 and implemented in the function `classify`.

```
R> classify <- function (data) {
+   clv <- data$clv
+   lora <- data$lora
+   cs <- data$cs
+   res <- ifelse(
+     (!is.na(clv) & !is.na(lora) & clv >= 5.1 &
+       lora >= 49.23372) |
+     (!is.na(clv) & !is.na(lora) & !is.na(cs) &
+       clv < 5.1 & lora >= 58.55409 &
+       cs < 1.405) |
+     (is.na(clv) & !is.na(lora) & !is.na(cs)
+       & lora >= 58.55409 & cs < 1.405) |
+     (!is.na(clv) & is.na(lora) & cs < 1.405),
```

```
+     0, 1)
+   factor(res, labels = c("normal", "glaucoma"))
+ }
```

Prediction of future observations is now performed by

```
R> predict(object = fit, cFUN = classify,
+   newdata = study.group[c(1:3, 86:88),])
[1] normal   normal   normal
               glaucoma glaucoma glaucoma
```

We execute a bagged indirect classification approach by choosing `pFUN = bagging` and specifying the number of bootstrap samples (Peters et al., 2002). Regression or classification trees are fitted for each bootstrap sample, with respect to the measurement scale of the specified intermediate variables

```
R> fit <- inclass(formula.indirect,
+   pFUN = bagging, nbagg = 50,
+   data = study.groupI)
R> fit

Indirect classification, with 3
           intermediate variables:
lora cs clv

Predictive model per intermediate
    is bagging with 50 bootstrap replications
```

The call for the prediction of values remains unchanged.

Error rate estimation

Classification rules are usually assessed by their misclassification rate. Hence, error rate estimation is of main importance. The function `errorest` implements a unified interface to several resampling based estimators. Referring to the example, we apply a linear discriminant analysis and specify the error rate estimator by `estimator = "cv"`, `"boot"` or `"632plus"`, respectively. A 10-fold cross validation is performed by choosing `estimator = "cv"` and `est.para = list(k = 10)`. The options `estimator = "boot"` or `estimator = "632plus"` deliver a bootstrap estimator and its bias corrected version *.632+* (see Efron and Tibshirani, 1997), we specify the number of bootstrap samples to be drawn by `est.para = list(nboot = 50)`. Further arguments are required to particularize the classification technique. The argument `predict` represents the chosen predictive function. For a unified interface `predict` has to be based on the arguments `object` and `newdata` only, therefore a wrapper function `mypredict` is necessary for classifiers which require more than those arguments or do not return the predicted classes by default. For a linear discriminant analysis with `lda`, we need to specify

```
R> mypredict.lda <- function(object, newdata){
+    predict(object, newdata = newdata)$class}
```

and calculate a 10-fold-cross-validated error rate estimator for a linear discriminant analysis by calling

```
R> errorest(diagnosis ~ ., data= study.group,
+   model=lda, estimator = "cv",
+   predict= mypredict.lda)

10-fold cross-validation estimator
    of misclassification error

Data:  diagnosis on .

Error 0.2675
```

For the indirect approach the specification of the call becomes slightly more complicated. Again for a unified interface a wrapper function has to be used, which incorporates the fixed classification rule

```
R> mypredict.inclass <-
+   function(object, newdata){
+   predict.inclass(object = object,
+     cFUN = classify, newdata = newdata)
+  }
```

The bias corrected estimator *.632+* is computed by

```
R> errorest(formula.indirect,
+   data = study.groupI, model = inclass,
+   predict = mypredict.inclass,
+   estimator = "632plus",
+   iclass = "diagnosis", pFUN = lm)

.632+ Bootstrap estimator of misclassification
  error with 25 bootstrap replications

Data:  diagnosis

Error 0.2658
```

Because of the subdivision of variables and a formula describing the modeling between explanatory and intermediate variables only, we must call the class membership variable. Hence, in contrast to the function `inclass` the data set `study.groupI` used in `errorest` must contain explanatory, intermediate and response variables.

To summarize the performance of different classification techniques in the considered example of glaucoma diagnosis, the 10-fold cross-validated error estimator delivers the results given in the following table:

method	error estimate
lda	0.2237
rpart	0.2529
bagging	0.1882
double-bagging	0.1941
inclass-bagging	0.2059
inclass-lm	0.2294

lda denotes the linear discriminant analysis, *rpart* a classification tree, *bagging* bagging with 50 bootstrap samples, *double-bagging* bagging with 50 bootstrap samples, combined with LDA, *inclass-bagging* indirect classification using bagging and *inclass-lm* indirect classification using linear modeling.

Note that an estimator of the variance is available for the ordinary bootstrap estimator (`estimator="boot"`) only, see Efron and Tibshirani (1997).

Summary

ipred tries to implement a unified interface to some recent developments in classification and error rate estimation. It is by no means finished nor perfect and we very much appreciate comments, suggestions and criticism. Currently, the major drawback is speed. Calling `rpart` 50 times for each bootstrap sample is relatively inefficient but the design of interfaces was our main focus instead of optimization. Beside the examples shown, `bagging` can be used to compute bagged regression trees and `errorest` computes estimators of the mean squared error for regression models.

Bibliography

L. Breiman. Bagging predictors. *Machine Learning*, 24(2): 123–140, 1996a. 33

Leo Breiman. Out-of-bag estimation. Technical report, Statistics Department, University of California Berkeley, Berkeley CA 94708, 1996b. 33, 34

B. Efron and R. Tibshirani. Improvements on cross-validation: The .632+ bootstrap method. *Journal of the American Statistical Association*, 92(438):548–560, 1997. 33, 35, 36

D.J. Hand, H.G. Li, and N.M. Adams. Supervised classification with structured class definitions. *Computational Statistics & Data Analysis*, 36:209–225, 2001. 33, 34

T. Hothorn and B. Lausen. Double-bagging: Combining classifiers by bootstrap aggregation. submitted, 2002. preprint available under http://www.mathpreprints.com/. 34

T. Hothorn, I. Pal, O. Gefeller, B. Lausen, G. Michelson, and D. Paulus. Automated classification of optic nerve head topography images for glaucoma screening. In *Studies in Classification, Data Analysis, and Knowledge Organization (to appear)*. Proceedings of the 25th Annual Conference of the German Classification Society, 2002. 33

A. Peters, T. Hothorn, and B. Lausen. Glaucoma diagnosis by indirect classifiers. In *Studies in Classification, Data Analysis, and Knowledge Organization (to appear)*. Proceedings of the 8th Conference of the International Federation of Classification Societies, 2002. 33, 35

Friedrich-Alexander-Universität Erlangen-Nürnberg, Institut für Medizininformatik, Biometrie und Epidemiologie, Waldstraße 6, D-91054 Erlangen
Andrea.Peters@imbe.imed.uni-erlangen.de
Torsten.Hothorn@rzmail.uni-erlangen.de
Berthold.Lausen@rzmail.uni-erlangen.de

Support from Deutsche Forschungsgemeinschaft SFB 539-C1/A4 is gratefully acknowledged.

Changes in R

by the R Core Team

User-visible changes

- XDR support is now guaranteed to be available, so the default save format will always be XDR binary files, and it is safe to distribute data in that format. (We are unaware of any platform that did not support XDR in recent versions of R.)

  `gzfile()` is guaranteed to be available, so the preferred method to distribute sizeable data objects is now via `save(compress = TRUE)`.

- `pie()` replaces `piechart()` and defaults to using pastel colours.

- `formatC()` has new arguments (see below) and `formatC(*, d = <dig>)` is no longer valid and must be written as `formatC(*, digits = <dig>)`.

- Missingness of character strings is treated much more consistently, and the character string `"NA"` can be used as a non-missing value.

- `summary.factor()` now uses a stable sort, so the output will change where there are ties in the frequencies.

New features

- Changes in handling missing character strings:

  - `"NA"` is no longer automatically coerced to a missing value for a character string. Use `as.character(NA)` where a missing value is required, and test via `is.na(x)`, not x

== "NA". String "NA" is still converted to missing by `scan()` and `read.table()` unless 'na.strings' is changed from the default.

  - A missing character string is now printed as 'NA' (no quotes) amongst quoted character strings, and '<NA>' if amongst unquoted character strings.
  - `axis()` and `text.default()` omit missing values of their 'labels' argument (rather than plotting "NA").
  - Missing character strings are treated as missing much more consistently, e.g., in logical comparisons and in sorts. `identical()` now differentiates "NA" from the missing string.

- Changes in package **methods**:
  - New function `validSlotNames()`.
  - Classes can explicitly have a "data part", formally represented as a `.Data` slot in the class definition, but implemented consistently with informal structures. While the implementation is different, the user-level behavior largely follows the discussion in *Programming with Data*.
  - A "next method" facility has been provided, via the function `callNextMethod()`. This calls the method that would have been selected if the currently active method didn't exist. See `?callNextMethod`. This is an extension to the API.
  - Classes can have initialize methods, which will be called when the function `new()` is used to create an object from the class. See `?initialize`. This is an extension to the API.
  - The logic of `setGeneric()` has been clarified, simplifying nonstandard generic functions and default methods.

- Changes in package **tcltk**:
  - Now works with the GNOME user interface.
  - Several new functions allow access to C level Tcl objects. These are implemented using a new 'tclObj' class, and this is now the class of the return value from `.Tcl()` and `tkcmd()`.

- Changes in package **ts**:
  - More emphasis on handling time series with missing values where possible, for example in `acf()` and in the ARIMA-fitting functions.
  - New function `arima()` which will replace `arima0()` in due course. Meanwhile, `arima0()` has been enhanced in several ways. Missing values are accepted. Parameter values can be initialized and can held fixed during fitting. There is a new argument 'method' giving the option to use conditional-sum-of-squares estimation.
  - New function `arima.sim()`.
  - New datasets AirPassengers, Nile, UKgas and WWWusage, and a expanded version of UKDriverDeaths (as a multiple time series Seatbelts).
  - New generic function `tsdiag()` and methods for arima and arima0, to produce diagnostic plots. Supersedes `arima0.diag()`.
  - New functions `ARMAacf()` and `ARMAtoMA()` to compute theoretical quantities for an ARMA process.
  - New function `acf2AR()` to compute the AR process with a given autocorrelation function.
  - New function `StructTS()` to fit structural time series, and new generic function `tsSmooth()` for fixed-interval state-space smoothing of such models.
  - New function `monthplot()` (contributed by Duncan Murdoch).
  - New functions `decompose()` and `HoltWinters()` (contributed by David Meyer) for classical seasonal decomposition and exponentially-weighted forecasting.

- An extensible approach to safe prediction for models with e.g. `poly()`, `bs()` or `ns()` terms, using the new generic function `makepredictcall()`. Used by most model-fitting functions including `lm()` and `glm()`. See `?poly`, `?cars` and `?ns` for examples.

- `acosh()`, `asinh()`, `atanh()` are guaranteed to be available.

- `axis()` now omits labels which are NA (but still draws the tick mark.

- Connections to bzip2-ed files via `bzfile()`.

- `chol()` allows pivoting via new argument 'pivot'.

- `cmdscale()` now takes rownames from a dist object 'd' as well as from a matrix; it has new arguments 'add' (as S) and 'x.ret'.

- `crossprod()` handles the case of real matrices with y = x separately (by accepting y = NULL). This gives a small performance gain (suggestion of Jonathan Rougier).

- `deriv()` and `deriv3()` can now handle expressions involving pnorm and dnorm (with a single argument), as in S-PLUS.

- New function `expm1()` both in R and in C API, for accurate $\exp(x) - 1$; precision improvement in `pexp()` and `pweibull()` in some cases. (PR#1334-5)

- New function `findInterval()` using new C entry point findInterval, see below.

- `formatDL()` now also works if both items and descriptions are given in a suitable list or matrix.

- `gzfile()` is guaranteed to be available, and hence the 'compress' option to `save()` and `save.image()`.

- `hist()` now has a method for date-time objects.

- `library()` now checks the dependence on R version (if any) and warns if the package was built under a later version of R.

- `library(help = PKG)` now also returns the information about the package PKG.

- Added function `logb()`, same as `log()` but for S-PLUS compatibility (where log now has only one argument).

- New na.action function `na.pass()` passes through NAs unaltered.

- `piechart()` has been renamed to `pie()`, as piechart is a Trellis function for arrays of pie charts. The default fill colours are now a set of pastel shades, rather than `par("bg")`.

- `plclust()` in package **mva**, for more S-PLUS compatibility.

- `poly()` now works with more than one vector or a matrix as input, and has a predict method for objects created from a single vector.

- `polyroot()` now handles coefficient vectors with terminal zeroes (as in S).

- New `prettyNum()` function used in `formatC()` and `format.default()` which have new optional arguments 'big.mark', 'big.interval', 'small.mark', 'small.interval', and 'decimal.mark'.

- `print.coefmat()` has a new argument 'eps.Pvalue' for determining when small P-values should be printed as '< {...}'.

- The `recover()` function has been moved to the **base** package. This is an interactive debugging function, usually a good choice for `options(error=)`. See `?recover`.

- `rep()` has a new argument 'each' for S-PLUS compatibility. The internal call is made available as `rep.int()`, again for help in porting code.

- New functions `rowSums()`, `colSums()`, `rowMeans()` and `colMeans()`: versions of `apply()` optimized for these cases.

- `rug()` now has a '`...`' argument allowing its location to be specified.

- `scan()` can have `NULL` elements in 'what', useful to save space when columns need to be discarded.

- New option 'by = "DSTday"' for `seq.POSIXt()`.

- Changes to sorting:

  - `sort()`, `sort.list()` and `order()` have a new argument 'decreasing' to allow the order to be reversed whilst still preserving ties.
  - `sort()` has an option to use quicksort in some cases (currently numeric vectors and increasing order).
  - The default Shell sort is Sedgewick's variant, around 20% faster, and pre-screening for NAs speeds cases without any NAs several-fold.
  - `sort.list()` (and order with just one vector) is several times faster for numeric, integer and logical vectors, and faster for character vectors.

- New assignment forms of `split()`; new function `unsplit()`.

- New `sprintf()` function for general C like formatting, from Jonathan Rougier.

- Argument 'split' of both `summary.aov` and `summary.aovlist` is now implemented.

- `summary.princomp()` now has a separate print method, and 'digits' is now an argument to the print method and not to `summary.princomp` itself.

- An extended version of the `trace()` function is available, compatible with the function in S-PLUS. Calls to R functions can be inserted on entry, on exit, and before any subexpressions. Calls to `browser()` and `recover()` are useful. See `?trace`.

- New function `TukeyHSD()` for multiple comparisons in the results of `aov()`. (Formerly function `Tukey` in package **Devore5** by Douglas Bates.)

- New read-only connections to files in zip files via `unz()`.

- `warning()` has new argument 'call.', like `stop()`'s.

- `zip.file.extract()` is no longer provisional and has an "internal" method available on all platforms.

- Methods for `[`, `[<-` and `as.data.frame()` for class `"POSIXlt"`.

- Much improved printing of matrices and arrays of type `"list"`.

- The "Knuth-TAOCP" option for random number generation has been given an option of using the 2002 revision. See ?RNG for the details: the R usage already protected against the reported 'weakness'.

- min/max of `integer(0)` (or `NULL`) is now `Inf`/`-Inf`, not an extreme integer.

Deprecated & defunct

- `.Alias()`, `reshapeLong()`, `reshapeWide()` are defunct.

- `arima0.diag()` (package **ts**) is deprecated: use `tsdiag()` instead.

- `piechart()` is deprecated; renamed to `pie()`.

Documentation changes

- *Writing R Extensions* now has an example of calling R's random numbers from FORTRAN via C.

- R itself and all R manuals now have ISBN numbers, please use them when citing R or one of the manuals.

Installation changes

- The configure script used when building R from source under Unix is now generated using Autoconf 2.50 or later, which has the following 'visible' consequences:

  - By default, configure no longer uses a cache file. Use the command line option '–config-cache' (or '-C') to enable caching.

  - Key configuration variables such as `CC` are now *precious*, implying that the variables

    * no longer need to be exported to the environment and can and should be set as command line arguments;

    * are kept in the cache even if not specified on the command line, and checked for consistency between two configure runs (provided that caching is used, see above);

    * are kept during automatic reconfiguration as if having been passed as command line arguments, even if no cache is used.

  See the variable output section of 'configure –help' for a list of all these variables.

- Configure variable `FC` is deprecated, and options '–with-g77', '–with-f77' and '–with-f2c' are defunct. Use configure variable `F77` to specify the FORTRAN 77 compiler, and `F2C` to specify the FORTRAN-to-C compiler and/or that it should be used even if a FORTRAN 77 compiler is available.

- Non-standard directories containing libraries are specified using configure variable `LDFLAGS` (not `LIBS`).

Utilities

- `Sweave()`, `Stangle()` and friends in package **tools**. Sweave allows mixing LaTeX documentation and R code in a single source file: the R code can be replaced by its output (text, figures) to allow automatic report generation. Sweave files found in package subdir 'inst/doc' are automatically tested by `R CMD check` and converted to PDF by `R CMD build`, see the section on package vignettes in *Writing R Extensions*.

- Rdconv can convert to the S4 '.sgml' format.

- 'R::Utils.pm' masks some platform dependencies in Perl code by providing global variables like `R_OSTYPE` or wrapper functions like `R_runR()`.

- If a directory 'inst/doc' is present in the sources of a package, the HTML index of the installed package has a link to the respective subdirectory.

- `R CMD check` is more stringent: it now also fails on malformed 'Depends' and 'Maintainer' fields in 'DESCRIPTION' files, and on unbalanced braces in Rd files. It now also provides pointers to documentation for problems it reports.

- `R CMD check`, `build` and `INSTALL` produce outline-type output.

- QA functions in package **tools** now return the results of their computations as objects with suitable `print()` methods. By default, output is only produced if a problem was found.

- New utility `R CMD config` to get the values of basic R configure variables, or the header and library flags necessary for linking against R.

- `Rdindex` and 'maketitle.pl' require Perl 5.005, as 'Text::Wrap::fill' was only introduced at 5.004_05.

C-level facilities

- All the double-precision BLAS routines are now available, and package writers are encouraged not to include their own (so enhanced ones will be used if requested at configuration).

- `findInterval(xt[],n,x,...)` gives the index (or interval number) of `x` in the sorted sequence `xt[]`. There's an `F77_SUB(interv)(.)` to be called from FORTRAN; this used to be part of `predict.smooth.spline`'s underlying FORTRAN code.

- Substitutes for (v)snprintf will be used if the OS does not supply one, so tests for `HAVE_(V)`SNPRINTF are no longer needed.

- The DUP and NAOK arguments in a `.C()` call are not passed on to the native routine being invoked. Any code that relied on the old behaviour will need to be modified.

- `log1p` is only provided in 'Rmath.h' if it is not provided by the platform, in which case its name is not remapped, but a back-compatibility entry point `Rf_log1p` is provided. Applications using libRmath may need to be re-compiled.

- The methods used by `integrate()` and `optim()` have entry points in 'R_ext/Applic.h' and have a more general interface documented in *Writing R Extensions*.

- The bessel_? entry points are now suitable to be called repeatedly from code loaded by `.C()`. (They did not free memory until `.C()` returned in earlier versions of R.)

- Server sockets on non-Windows platforms now set the SO_REUSEADDR socket option. This allows a server to create simultanous connections to several clients.

- New quicksort sorting (for numeric no-NA data), accessible from C as `R_qsort()` etc and from FORTRAN as `qsort4()` and `qsort3()`.

- 'Rinternals.h' no longer includes 'fcntl.h', as this is not an ISO C header and cannot be guaranteed to exist.

- FORTRAN subroutines are more correctly declared as 'extern void' in 'R_exts/Applic.h' and 'R_exts/Linpack.h'.

Bug fixes

- The calculation of which axes to label on a `persp()` plot was incorrect in some cases.

- Insufficient information was being recorded in the display list for the `identify()` function. In particular, the 'plot' argument was ignored when replaying the display list. (PR#1157)

- The vertical alignment of mathematical annotations was wrong. When a vertical adjustment was not given, it was bottom-adjusting i.e,. it was treating `adj=0` as `adj=c(0, 0)`. It now treats `adj=0` as `adj=c(0, 0.5)` as for "normal" text. (PR#1302)

- the man page ('doc/R.1') wasn't updated with the proper R version.

- `smooth.spline()` had a 'df = 5' default which was never used and hence extraneous and misleading.

- `read.fwf()` was interpreting comment chars in its call to scan: replaced by a call to `readLines()`. (PR#1297/8)

- The default comment char in `scan()` has been changed to '`""`' for consistency with earlier code (as in the previous item).

- `bxp(*, notch.frac = f)` now draws the median line correctly.

- Current versions of `gs` were rotating the output of `bitmap(type = "pdfwrite")` and when converting the output of `postscript()` to PDF; this has been circumvented by suppressing the '%%Orientation' comment for non-standard paper sizes.

- `plot.ts(x, log = "y")` works again when `x` has 0s, also for matrix `x`.

- `add1()`, `drop1()`, `step()` work again on glm objects with formulae with rhs's containing '.'. (Broken by a 'bug fix' (in reality an API change) in 1.2.1.)

- `optim(method="BFGS")` was not reporting reaching 'maxit' iterations in the convergence component of the return value.

- `aov()` and `model.tables()` were failing on multistrata models with excessively long Error formula. (PR#1315)

- Transparent backgrounds on `png()` devices on Unix-alikes had been broken during the driver changes just prior to 1.4.0. (They worked correctly on Windows.)

- `demo(is.things)` didn't work properly when the methods package was attached.

- `match()`,`unique()` and `duplicated()` were not declaring all `NaN`s to be equal, yet not always distinguishing `NA` and `NaN`. This was very rare except for data imported as binary numbers.

- The error handler `recover()` protects itself against errors in dump.frames and uses a new utility, limitedLabels, to generate names for the dump that don't inadvertently blow the limit on symbol length. (TODO: either fix dump.frames accordingly or remove the limit– say by truncating very long symbols?)

- `se.contrasts()` works more reliably with multistratum models, and its help page has an example.

- `summary.lm()` was not returning r.squared nor adj.r.squared for intercept-only models, but `summary.lm.null()` was returning r.squared but not adj.r.squared. Now both are always returned. Neither returned f.statistic, and that is now documented.

- Subsetting of matrices of mode `"list"` (or other non-atomic modes) was not implemented and gave incorrect results without warning. (PR#1329). Under some circumstances subsetting of a character matrix inserted `NA` in the wrong place.

- `abs()` was not being treated as member of the Math group generic function, so e.g. its method for data frames was not being used.

- `set.seed(seed, "default")` was not using the 'seed' value (only for 'kind = "default"').

- `logLik.lm()` now uses 'df = p + 1' again ('+ sigma'!).

- `logLik.glm()` was incorrect for families with estimated dispersion.

- Added `strptime()` workaround for those platforms (such as Solaris) that returned missing components as 0. Missing days are now detected, but missing years will still be interpreted as 1900 on such platforms.

- Inheritance in formal classes (the **methods** package) works breadth-first as intuition would expect.

- The `new()` function in package **methods** works better (maybe even correctly?) for the various combinations of super-classes and prototypes that can be supplied as unnamed arguments.

- Internal code allowed one more connection to be allocated than the table size, leading to segfaults. (PR#1333)

- If a user asks to open a connection when it is created and it cannot be opened, the connection is destroyed before returning from the creation call. (related to PR#1333)

- `Sys.putenv()` was not using permanent storage. (PR#1371)

- `La.svd()` was not coercing integer matrices. (PR#1363)

- `deriv(3)` now reports correctly the function it cannot find the derivatives table.

- The GNOME user interface was over-enthusiastic about setting locale information. Now only `LC_CTYPE`, `LC_COLLATE` and `LC_TIME` are determined by the user's environment variables (PR#1321).

- In X11, `locator()` would sound the bell even if `xset b off` had been set.

- `merge()` could be confused by inconsistent use of `as.character()` giving leading spaces.

- `[pqr]binom()` no longer silently round the 'size' argument, but return `NaN` (as `dbinom()` does). (PR#1377)

- Fixed socket writing code to block until all data is written. Fixed socket reading code to properly handle long reads and reads with part of the data in the connection buffer.

- Allow sockets to be opened in binary mode with both 'open="ab"' and 'open="a+b"'.

- `levels<-.factor()` was using incorrectly list values longer than the number of levels (PR#1394), and incorrectly documented that a character value could not be longer than the existing levels.

- The `pdf()` device was running out of objects before the documented 500 page limit. Now there is no limit.

- `legend()` did not deal correctly with 'angle' arguments. (PR#1404)

- `sum()` tried to give an integer result for integer arguments, but (PR#1408)

  - this was not documented
  - it sometimes warned on overflow, sometimes not
  - it was order-dependent for a mixture of integer and numeric args.

- `mean()` gave (numeric) `NA` if integer overflow occurred in `sum()`, but now always works internally with numeric (or complex) numbers.

- `sort.list()` and `order()` were treating NA_STRING as "NA".

- `sort.list(na.last = NA)` was not implemented.

- `seq.default()` was returning only one element for a relative range of less than about 1e-8, which was excessively conservative. (PR#1416)

- `tsp(x) <- NULL` now also works after attaching the **methods** package.

- `persp(shade=)` was not working correctly with the default col=NULL if this was transparent. (PR#1419)

- `min(complex(0))` and `max(complex(0))` were returning random values.

- `range()` gave `c(1, 1)`.

- `range(numeric(0))` is now `c(Inf, -Inf)`, as it was documented to be.

- `print.ts()` was occasionally making rounding errors in the labels for multiple calendar time series.

- `Rdconv` was not handling nested `\describe{}` constructs in conversion to HTML (PR#1257) and not fixing up mal-formed `\item` fields in `\describe{}` in conversion to text (PR#1330).

- `filled.contour()` was not checking consistency of x, y, z. (PR#1432)

- `persp.default()` no longer crashes with non-character labels. (PR#1431)

- `fft()` gave incorrect answers for input sizes 392, 588, 968, 980, … (PR#1429)

- `det(method = "qr")` gave incorrect results for numerically singular matrices. (PR#1244)

- `barplot()` now allows the user to control 'xpd'. (PR#1088, 1398)

- `library()` (with no arguments) no longer fails on empty '`TITLE`' files.

- `glm()` was failing if both `offset()` and start were specified. (PR#1421)

- `glm()` might have gotten confused if both step-shortening and pivoting had occurred (PR#1331). Step-halving to avoid the boundary of feasible values was not working.

- Internal representation of logical values was not being treated consistently. (Related to PR#1439)

- The `c()` function sometimes inserted garbage in the name vector for some types of objects, e.g. `names(c(ls, a=1))`.

- Fixed bug in '`$`' that could cause mutations on assignment (PR#1450).

- Some X servers displayed random bytes in the window title of graphics windows (PR#1451)

- The X11 data editor would segfault if closed with window manager controls (PR#1453)

- Interrupt of `Sys.sleep()` on UNIX no longer causes subsequent `Sys.sleep()` calls to segfault due to infinite recusion.

- Eliminated a race condition that could cause segfaults when a SIGINT was received while handling an earlier SIGINT.

- `rect(lty = "blank")` was incorrectly drawing with a dashed line.

- `type.convert()` was not reporting incorrectly formatted complex inputs. (PR#1477)

- `readChar()` was not resetting vmax, so causing memory build-up. (PR#1483)

Changes on CRAN

by Kurt Hornik

CRAN packages

DBI A common database interface (DBI) class and method definitions. All classes in this package are virtual and need to be extended by the various DBMS implementatios. By the R Special Interest Group on Databases (R-SIG-DB).

Rmpi Rmpi provides an interface (wrapper) to MPI APIs. It also provides interactive R slave functionalities to make MPI programming easier in R than in C(++) or FORTRAN. By Hao Yu.

VLMC Functions, classes & methods for estimation, prediction, and simulation (bootstrap) of VLMC – Variable Length Markov Chain – Models. By Martin Maechler.

brlr Fits logistic regression models by maximum penalized likelihood. By David Firth.

cobs Qualitatively Constrained (Regression) Smoothing via Linear Programming. By Pin T. Ng and Xuming He, U. Illinois; R port by Martin Maechler.

dblcens Use EM algorithm to compute the NPMLE of CDF and also the two censoring distributions. Data can be doubly censored. You can also specify a constraint, it will return the constrained NPMLE and the -2 log likelihood ratio. This can be used to test the hypothesis and find confidence interval for F(K) via empirical likelihood ratio theorem. Influence function may be calculated (but slow). By Mai Zhou, Li Lee, Kun Chen.

dichromat Collapse red-green distinctions to simulate the effects of colour-blindness. By Thomas Lumley.

gllm Routines for log-linear models of incomplete contingency tables, including some latent class models via EM and Fisher scoring approaches. By David Duffy.

gtkDevice GTK graphics device driver that may be used independently of the R-GNOME interface and can be used to create R devices as embedded components in a GUI using a Gtk drawing area widget, e.g., using RGtk. By Lyndon Drake; packaging and extensions by Martyn Plummer and Duncan Temple Lang.

knnTree Construct or predict with k-nearest-neighbor classifiers, using cross-validation to select k, choose variables (by forward or backwards selection), and choose scaling (from among no scaling, scaling each column by its SD, or scaling each column by its MAD). The finished classifier will consist of a classification tree with one such k-nn classifier in each leaf. By Sam Buttrey.

ipred Improved predictive models by direct and indirect bootstrap aggregation in classification and regression as well as resampling based estimators of prediction error. By Andrea Peters and Torsten Hothorn.

npmc Provides simultaneous rank test procedures for the one-way layout without presuming a certain distribution. By Joerg Helms, Ullrich Munzel.

randomForest Classification based on a forest of classification trees using random inputs. FORTRAN original by Leo Breiman and Adele Cutler, R port by Andy Liaw and Matthew Wiener.

rsprng Provides interface to SPRNG APIs, and examples and documentation for its use. By Na (Michael) Li.

serialize Simple interfce for serializing to connections. By Luke Tierney.

spdep A collection of functions to create spatial weights matrix objects from polygon contiguities, from point patterns by distance and tesselations, for summarising these objects, and for permitting their use in spatial data analysis; a collection of tests for spatial autocorrelation, including global Moran's I and Geary's C, local Moran's I, saddlepoint approximations for global and local Moran's I; and functions for estimating spatial simultaneous autoregressive (SAR) models. Was formerly the three packages: spweights, sptests, and spsarlm. By Roger Bivand, with contributions by Nicholas Lewin-Koh and Michael Tiefelsdorf.

subselect A collection of functions which assess the quality of variable subsets as surrogates for a full data set, and search for subsets which are optimal under various criteria. By Jorge Orestes Cerdeira, Jorge Cadima and Manuel Minhoto.

systemfit This package contains functions for fitting simultaneous systems of equations using Ordinary Least Sqaures (OLS), Two-Stage Least Squares (2SLS), and Three-Stage Least Squares (3SLS). By Jeff D. Hamann.

tkrplot simple mechanism for placing R graphics in a Tk widget. By Luke Tierney.

CRAN mirrors the R packages from the Omegahat project in directory 'src/contrib/Omegahat'. The following is a recent addition:

RGtkViewers GUI tools for viewing databases, S4 class hierarchies, etc. By Duncan Temple Lang.

Kurt Hornik
Wirtschaftsuniversität Wien, Austria
Technische Universität Wien, Austria
Kurt.Hornik@R-project.org

Upcoming Events

by Kurt Hornik and Friedrich Leisch

JSM 2002

The Joint Statistical Meetings taking place in New York on August 11–15, 2002 will feature a number of R-related activities. Paul Murrell will chair a session on *"R Graphics"* with talks on R graphics desiderata (Vincent Carey), scatterplot3d (Uwe Ligges), an R interface to OpenGL (Duncan Murdoch), embedding R graphics in Excel (Erich Neuwirth) and GGobi & R (Deborah Swayne). In addition, Paul will give a talk on using color in graphs.

Brian Yandell has organized a session entitled *"The Future of Electronic Publication: Show Me ALL the Data"*. It consists of talks on extensible formats for data analysis & documentation (Friedrich Leisch), analysis of microarray data (Robert Gentleman), strategies for software quality assurance (Kurt Hornik) and dynamic statistical documents (Duncan Temple Lang).

DSC 2003

The third international workshop on 'Distributed Statistical Computing' (DSC 2003) will take place at the Technische Universität Wien in Vienna, Austria from 2003-03-19 to 2003-03-21. This workshop will deal with future directions in statistical computing and graphics.

Particular emphasis will be given to R and open-source projects related R, including Omegahat (http://www.omegahat.org/) and BioConductor (http://www.bioconductor.org/). DSC 2003 builds on the spirit and success of DSC 1999 and DSC 2001, which were seminal to the further development of R and Omegahat.

This should be an exciting meeting for everyone interested in statistical computing with R.

The conference home page at http://www.ci.tuwien.ac.at/Conferences/DSC-2003/ gives more information.

Kurt Hornik
Wirtschaftsuniversität Wien, Austria
Technische Universität Wien, Austria
Kurt.Hornik@R-project.org

Friedrich Leisch
Technische Universität Wien, Austria
Friedrich.Leisch@R-project.org

Editor-in-Chief:
Kurt Hornik
Institut für Statistik und Wahrscheinlichkeitstheorie
Technische Universität Wien
Wiedner Hauptstraße 8-10/1071
A-1040 Wien, Austria

Editorial Board:
Friedrich Leisch and Thomas Lumley.

Editor Programmer's Niche:
Bill Venables

R News is a publication of the R project for statistical computing, communications regarding this publication should be addressed to the editors. All articles are copyrighted by the respective authors. Please send submissions to the programmer's niche column to Bill Venables, all other submissions to Kurt Hornik, Friedrich Leisch, or Thomas Lumley (more detailed submission instructions can be found on the R homepage).

R Project Homepage:
http://www.R-project.org/

Email of editors and editorial board:
firstname.lastname@R-project.org

This newsletter is available online at
http://CRAN.R-project.org/doc/Rnews/