R News

The Newsletter of the R Project — Volume 1/3, September 2001

Editorial

by Friedrich Leisch

While our colleagues in the southern hemisphere are looking forward to the last weeks of the academic year and a well deserved summer break, teaching started again in the northern half of our world, and again we had a new bug-fix release of R ready to pass to students at the beginning of the semester.

Coffee break talks at this summer's conference season showed the growing popularity of R at universities very nicely to me: You start talking to people you never met before and one of the first questions is usually about each other's research interests. Until last year I always had to explain what R is after responding "Hacking R", because nine out of ten people at a statistics conference had never heard about it.

This year things were different. I participated in a conference on Mixture Models in Hamburg, Germany. The crowd there were mostly statisticians, using computers heavily for their work, but most of them are not interested in statistical computing *per se* as a research area. Nevertheless, every other one at least knew what R is, many use it actively or at least their students do so, several talks mentioned R as the computational platform used for the examples (and in several others design and layout of graphics looked *very* familiar to me).

This issue of *R News* has both articles introducing R packages and articles dealing with more computational aspects: The former include articles on econometrics, spatial data, machine learning, and robust statistics; the latter articles on Mac OS X, parallel and object-oriented programming, dynamic access to compiled code, and graphical user interfaces. The programmer's niche column is guest-edited by Thomas Lumley and deals with the advantages and disadvantages of macros.

The next issue of *R News* scheduled for the end of 2001 will have an emphasis on applying R in medical statistics. In the future we would like to see one or two issues per year to have a main focus, where 3 or 4 articles deal with related topics. Suggestions for focus topics are welcome, especially if accompanied by prospective articles. Please contact the editors for details.

Friedrich Leisch
Technische Universität Wien, Austria
Friedrich.Leisch@ci.tuwien.ac.at

Contents of this issue:

Porting R to Darwin/X11 and Mac OS X

by Jan de Leeuw

Mac OS X

Earlier this year Apple officially released OS X, its new operating system. OS X now comes pre-installed on all Macs, although by default you still boot into MacOS 9.x. But soon OS X will be the default.

OS X is not an incremental upgrade, it is a completely new operating system. It has a layered architecture. The lowest layer is Darwin, which consists of the Mach 3.0 kernel and a version of BSD 4.4. Thus OS X is, among other things, a certified and POSIX compliant Unix. Darwin is (certified) Open Source, and it can be downloaded from the Apple servers. One surprising consequence of the above is that soon Apple will be the largest distributor of Unix, and that soon OS X will be the most popular Unix on the planet, although most users will be blissfully unaware of this fact.

On top of Darwin there is a lot of proprietary software, used to generate the user interface components. The main libraries are Apple's version of OpenGL for 3D, QuickTime for multimedia, and Quartz for printing and screen drawing. Quartz replaces Display Postscript in earlier versions of the system, using PDF as its native format.

Application environments

On top of the three graphics engines are no less than five application environments that developers can use.

Classic For the foreseeable future it will remain possible to boot into OS 9.x, and to run older Macintosh programs in the Classic environment with OS X, which emulates an older Macintosh with OS 9.x. Some of the more powerful programs for the Mac, such as Office and Photoshop and SPSS, still have to run in Classic, although Carbon versions have been announced.

Carbon The classical Mac Toolbox API has been cleaned up and extended. This now makes it possible to write applications that run natively on both OS 9.x and OS X. Netscape, MSIE, R, Stata, AppleWorks have all been carbonized. It is of some interest, however, that there are two types of Carbon applications. Those that can run on OS 9.x are under the control of the Code Fragment Manager and use the PEF executable format. If run on OS X, they run on top of a layer that translates CFM/PEF to dyld/Mach-O. Mach-O is the native format for OS X, and program control is exercised by the dynamic linker dyld. The other type of Carbon application is dyld/Mach-O, which means it does not run on OS 9.x.

Cocoa This is the native OS X API, inherited from its NeXTStep and Rhapsody parents and grandparents. Applications using these interfaces use optimally the capacities of the OS. Cocoa applications are still comparatively rare, because they have to be written from scratch, either in Objective-C or in Java. But there are already fine browsers, spreadsheets, editors, graphic tools, and TeX systems in Cocoa.

Java The JDK (including runtime, JIT compiler, AWT, and Swing) is integrated with OS X, and Java libraries are available to write Cocoa applications. Swing, of course, has the native OS X look-and-feel. Of course anything you write in Java on OS X is (at least in principle) completely portable.

BSD Darwin comes with optimized Apple versions of the GNU tools. Since the application environment for Darwin is FreeBSD, porting of Unix programs is a breeze. It can be made even easier by using Fink (see below). In particular, it is trivial to install an X server, in fact a complete X11R6, using Xfree86, and a large number of different window managers. There are ports to Darwin of all of gnome, including the Gimp and Guppi, of various Matlab like programs such as octave, scilab, yorick, and of all the standard X graphic tools such as xfig, tgif, xpdf, xdvi, xv, ghostview, gnuplot, grace, xgobi.

User experience

The Mac OS X user, of course, will not notice any of these under-the-hood changes. The obvious change is Aqua, the new look-and-feel, often described as "lickable". Windows and the menu bar look different, there is a "dock", and so on. The Aqua interface is automatic for all Cocoa and Carbon applications that use Quartz to draw to the screen.

The user will notice greatly increased stability of the OS. This is mostly provided by the Mach kernel, which provides protected and advanced virtual memory, as well as preemptive and cooperative multitasking and threading. OS X will run forever without crashing, and applications that crash don't take down the OS with them any more. The need to reboot has virtually disappeared.

Moreover, OS X promises speed, although not in the early versions. The OS is written to take full advantage of multiprocessing, and multiprocessor Macs are becoming more and more common. Rumor has it that the G5 will even be multicore. Many graphics programs, including Quartz and OpenGL, are optimized for the Altivec vector processor on the G4 chip. Recent builds of the OS show great speed.

Finally, remember that OS X is first and foremost a Unix, i.e. a multitasking and multiuser OS. You have to login, you can allow others to login, and people can login remotely. Although you can use the system as a dedicated single-person desktop OS, that is only one of its uses. There are many people who log into the Cube in my office.

Porting problems

Darwin/X11 programmers must take into account some important differences with the more usual ELF based Unix systems. Most of those are due to the Mach heritage. All these peculiarities had to be taken into account in building R, and in modifying the `autoconf` configure files.

In the first place, Darwin maintains a strict distinction between two types of shared libraries. There are bundles, which can be loaded at runtime into an application using the appropriate dynamic loading interface. Also, there are dynamic libraries, that are used at link time when building applications or other libraries. Different compiler and linker switches are needed to build the two different types of libraries. For ELF systems the two types coincide. Building R as a shared (dynamic) library, which can be linked against other application programs, will be available in R-1.4.0 and does not work yet in R-1.3.1. The modules and packages which use bundles of object code that are loaded at runtime work fine.

Second, the Darwin dynamic linker `dyld` is very intolerant, and does not allow multiply defined symbols at all. The static linker is much more tolerant. Thus one must make sure not to include a file with definitions more than once, and so on.

Third, the API for dynamic loading is very different from the more usual `dlopen()` interface in ELF systems.

And finally, some of the necessary components needed for building R (X11R6, a Fortran compiler) are missing from the current version of Darwin.

Fink

The task of porting BSD and X11 software has been made easy by the existence of Fink (see http://fink.sourceforge.net). This is a package management system for Darwin setup by Christoph Pfisterer, and maintained by a group of volunteers. There are now more than 300 packages in Fink, and you can say `fink install foo` to download, configure, compile, and install package foo, and then `fink update foo` to update the package when it has changed in the Fink central location. Of course such package management systems exist for Linux, Debian, FreeBSD (and actually for R and Stata), but it is good to have one for Darwin as well.

What do you need from Fink for building a Darwin version of R? In the first place `Xfree86`. The Darwin version has been modified with a Cocoa front end called XDarwin that let's you choose between full-screen and rootless mode, where in rootless mode the X11 windows exist on the same desktop as the Aqua windows of the OS X Finder. Second, you can install all of gnome, which can be used for the (experimental and unsupported) gnome module in R. Third, Fink has ATLAS, an optimized BLAS library for OS X. Fourth, there is `dlcompat`. This wraps the dyld API for dynamic loading in the familiar ELF `dlopen` API, so you can continue to use the standard calls in the R sources. Fifth, there is `tcl/tk`, for the `tcltk` package in R. And finally there are various other libraries, which are either not in Darwin or are more recent versions. Examples are `libjpeg`, `libpng`, `libz`, and `libreadline`. There is also a `g77` in Fink, but it does not work with the configure scripts in R, so all our builds so far use `f2c`.

In fact, R-1.3.1 base and recommended are both in Fink. The info scripts and patch files are maintained by Jeffrey Whitaker (jsw@cdc.noaa.gov). This provides you with yet another way to install R on your Mac.

R

Combining all this new knowledge makes it possible to describe what we have on CRAN and what we still need. We have a CFM/PEF Carbon version of R, made by Stefano Iacus, and described in the first issue of R-News. It uses a Carbon version of the Macintosh QuickDraw driver. We also have a Darwin/X11 version, with support for Tcl/Tk, GNOME, and ATLAS, maintained by Jan de Leeuw (me).

The Carbon version runs on both OS 9.x and OS X, but we have seen that it needs a dyld/Mach-O layer to run on OS X, so it's not really native. There is no support in the Carbon version for Tcl/Tk, and the internet-based R package update and install system is not available. There are no free tools to build this version in OS X; you have to build it in OS 9.x, or buy an IDE from Metrowerks or Absoft.

The Darwin/X11 version is dyld/Mach-O, and is consequently native in that sense, but it does not use the native Quartz library and Cocoa interfaces at all. If you run the X server in full-screen mode, your Mac looks just like a Linux or Solaris machine. This is somewhat disappointing for Mac people.

There are various ways in which the current situation can be improved. Stefano is working on a Quartz driver for the graphics. It would be useful to have a dyld/Mach-O Carbon version, truly native to OS X. The Quartz driver also brings us closer to a Cocoa version of R, which could be implemented initially as a Cocoa shell around the Darwin version of R.

Much will depend on the reception of OS X, and on how many Mac users will switch from 9.x to X. If your hardware supports OS X, I think switching is a no-brainer, especially if you program, develop, or compute. As I have indicated above, the possibilities are endless.

Jan de Leeuw
University of California at Los Angeles
deleeuw@stat.ucla.edu

RPVM: Cluster Statistical Computing in R

by Michael Na Li and A.J. Rossini

rpvm is a wrapper for the *Parallel Virtual Machine* (PVM) API. PVM (Geist et al., 1994) is one of the original APIs for extending an application over a set of processors in a parallel computer or over machines in a local area cluster. We discuss the PVM API, how it is implemented in R, and provide examples for its use. **rpvm** provides a quick means for prototyping parallel statistical applications as well as for providing a front-end for data analysis from legacy PVM applications.

Introduction

PVM was developed at Oak Ridge National Laboratories and the University of Tennessee starting in 1989. It is a *de facto* standard for distributed computing designed especially for heterogeneous networks of computers. The notion of "virtual machine" makes the network appear logically to the user as a single large parallel computer. It provides a mechanism for specifying the allocation of tasks to specific processors or machines, both at the start of the program as well as dynamically during runtime. There are routines for the two main types of intertask communication: point-to-point communication between tasks (including broadcasting) and collective communication within a group of tasks.

The primary message passing library competitor to PVM is MPI (*Message Passing Interface*). The biggest advantage of PVM over MPI is its flexibility (Geist et al., 1996). PVM can be run on an existing network consisting of different platforms (almost all platforms are supported, including Microsoft Windows 98/NT/2000 systems). Tasks can be dynamically spawned, which is not supported in MPI-1 upon which most MPI implementations are based. Hosts can be dynamically added or deleted from the virtual machine, providing fault tolerance. There are also a visualization tool, `xpvm`, and numerous debugging systems. MPI has advantages of speed as well as being an actual standard. However, for prototyping and research, it isn't clear that either of these are critical features.

PVM has been successfully applied to many applications, such as molecular dynamics, semiconductor device simulation, linear algebra (ScaLAPACK, NAG PVM library), etc. It also has great potential in statistical computing, including optimization (expensive or large number of function evaluations; likelihood computations), simulations (resampling, including bootstrap, jackknife, and MCMC algorithms; integration), enumeration (permutation and network algorithms), solution of systems of equations (linear, PDE, finite-element, CFD).

This article presents a new R package, **rpvm**, that provides an interface to PVM from one of the most powerful and flexible statistical programming environments. With **rpvm**, the R user can invoke either executable programs written in compiled language such as C, C++ or FORTRAN as child tasks or spawn separate R processes. It is also possible to spawn R processes from other programs such as Python, C, FORTRAN, or C++. Therefore **rpvm** is ideal for prototyping parallel statistical algorithms and for splitting up large memory problems. Using **rpvm**, statisticians will be able to prototype difficult statistical computations easily in parallel. The rest of the article which follows looks at installation, features, a programming example, and concludes with issues for on-going development.

Installation

PVM source code can be downloaded from http://www.netlib.org/pvm3/pvm3.4.3.tgz. Binary distributions exist for many Linux distributions (see individual distributions) as well as for Microsoft Windows NT/2000/XP. However, the Windows implementation of **rpvm** is untried (it is possible to communicate with C or FORTRAN processes running under Microsoft Windows). The following procedures refer to UNIX-like environments.

Installing PVM

Compiling the source code: Installation from the source is straightforward. After untarring the source package, set the environment variable PVM_ROOT to where pvm resides, for example '$HOME/pvm3' or '/usr/local/pvm3'. Then type 'make' under the '$PVM_ROOT' directory. The libraries and executables are installed in '$PVM_ROOT/lib/$PVM_ARCH', where PVM_ARCH is the host architecture name, e.g., 'LINUX' or 'SUN4SOL2'. This way one can build PVM for different architectures under the same source tree.

PVM comes with plenty of examples, see the PVM documentation on how to build and run these.

Setting up PVM environment: Before running PVM, some environment variables need to be set. For example, if you use a C shell, put the following in the '$HOME/.cshrc' file of each host,

```
setenv PVM_ROOT $HOME/pvm3
setenv PVM_ARCH ‘$PVM_ROOT/lib/pvmgetarch‘
set path = ( $path $PVM_ROOT/lib \
             $PVM_ROOT/lib/$PVM_ARCH \
             $PVM_ROOT/bin/$PVM_ARCH )
```

PVM uses rsh by default to initialize communication between hosts. To use ssh (Secure Shell) instead, which is necessary for many networks, define

```
setenv PVM_RSH ‘which ssh‘
```

You can use public key authentication to avoid typing passwords; see the SSH documentation on how to do this.

Setting up RPVM

rpvm uses a shell script '$R_LIBS/rpvm/slaveR.sh' to start a slave R process. After installing **rpvm**, copy this file to '$PVM_ROOT/bin/$PVM_ARCH' so it can be found by the pvm daemon. The path to the slave R script and the slave output file can either be specified through environment variables RSLAVEDIR, RSLAVEOUT or by passing corresponding arguments to the spawning function. The first method can be used when different paths are needed for different host. When the hosts use a shared file system, the second method provides more flexibility. If neither are set, their default values '$R_LIBS/rpvm' and '$TMPDIR' are used.

A sample RPVM session

Below is a sample **rpvm** session. We start the virtual machine by using a host file, '$HOME/.xpvm_hosts',

```
> library(rpvm)
> hostfile <-
+   file.path(Sys.getenv("HOME"), ".xpvm_hosts")
> .PVM.start.pvmd (hostfile)
libpvm [t40001]: pvm_addhosts():
  Already in progress
libpvm [t40001]: pvm_addhosts():
  Already in progress
[1] 0
> .PVM.config()
There are 2 hosts and 2 architectures.
  host.id    name     arch speed
1  262144  abacus    LINUX  1000
2  524288   atlas SUN4SOL2  1000
```

A host file is a simple text file specifying the host names of the computers to be added to the virtual machine. A simple example is shown below.

```
* ep=$HOME/bin/$PVM_ARCH
atlas
abacus
```

where * defines a global option for all hosts. ep=*option* tells the execution path in which we want pvm daemon to look for executables. For more information, please refer to the PVM documentation.

In directory '$R_LIBS/rpvm/demo', there is a test script 'pvm_test.R' which spawns itself as a slave and receives some messages from it.

```
> source(file.path(Sys.getenv("R_LIBS"),
   "rpvm", "demo", "pvm_test.R"))
## Spawning  1 children
### Spawned  1 Task, waiting for data
Message received from  262165
Hello World! from abacus
Some integers  10 7 13
Some doubles  11.7633 11.30661 10.45883
And a matrix
            [,1]        [,2]       [,3]
[1,] -0.76689970 -1.08892973 -0.1855262
[2,] -0.08824007  0.26769811 -1.1625034
[3,]  1.27764749  0.05790402 -1.0725616
Even a factor!
 [1] s t a t i s t i c s
Levels:  a c i s t
```

If this example fails, check to make sure that '$R_LIBS/rpvm/slaveR.sh' is in the executable search path of the pvm daemon and pvm is running.

Features

rpvm provides access to the low-level PVM API as well as to higher-level functions for passing complex R data types such as matrices and factors. Future development will work at extensions to lists and data frames as well as eventually to functions and closures.

Specifically, APIs are provided for the following tasks:

- Virtual Machine Control: to start the virtual machine, add and delete hosts, query the configuration of VM and nodes status, shut down the VM.

- Task Control: to enter and exit from pvm, to start and stop children tasks, query task running status, etc.
- Message Passing: to prepare and send message buffers, to receive message with or without blocking or with timeout, to pack and unpack data, etc.
- Miscellaneous functions to set and get pvm global options, etc.

The implementation currently lacks the functions for Task Grouping, which is planned for the next release.

rpvm also aims in the long run to provide some general purpose functionality for some "naturally" parallel problems (known as "embarrassingly" parallel to computer scientists), such as parallel "apply" (function `PVM.rapply` in the associated script 'slapply.R' being the first attempt) as well as common tasks such as simple Monte Carlo algorithms for bootstrapping.

Using RPVM

Strategies for parallel programming

One common approach to parallel program design (Buyya, 1999) is a master-slave paradigm where one of the tasks is designated the master task and the rest are slave tasks. In general, the master task is responsible for spawning the slave tasks, dividing and sending workload, collecting and combining results from the slaves. The slave tasks only participate in the computation being assigned. Depending on the algorithm, the slaves may or may not communicate among themselves. For PVM, the process is summarized as Master tasks:

- Register with PVM daemon.
- Spawn slaves.
- Send Data.
- Collect and combine results.
- Return and quit.

and Slave tasks:

- Register with PVM daemon.
- Locate parent.
- Receive Data.
- Compute.
- Send results
- Quit.

Alternatively, instead of a star-like topology, one might consider a tree-like process where each task decides if it should split sub-tasks (and later join) or compute and return. Each task is the master to its children and a slave to its parent. This strategy is natural for "divide and conquer" algorithms and a variant of the master-slave paradigm. This might look like:

- Register with PVM daemon
- Determine if I'm the parent or a spawned process.
- Receive data if spawned (already have data if parent).
- Determine if I compute, or if I let slaves compute.
- If slaves compute:
  - Spawn slaves.
  - Send data to slaves.
  - Receive data from slaves.
- Compute.
- If spawned, send results to parent.
- Quit.

This may involve more message passing overhead but may be more efficient for some problems or network architectures and topologies.

Example

`.PVM.rapply` implements a preliminary version of parallel apply function. It divides a matrix up by rows, sends the function to `apply` and the submatrices to slave tasks and collects the results at the end. It is assumed that the slave script knows how to evaluate the function and returns a scalar for each row.

```
PVM.rapply <-
function(X, FUN = mean, NTASK = 1) {
    ## arbitrary integers tag message intent
    WORKTAG  <- 22
    RESULTAG <- 33
    end   <- nrow(X)
    chunk <- end %/% NTASK + 1
    start <- 1
    ## Register process with pvm daemon
    mytid <- .PVM.mytid()
    ## Spawn R slave tasks
    children <- .PVM.spawnR(ntask = NTASK,
                            slave = "slapply")
    ## One might check if spawning successful,
    ## i.e. entries of children >= 0 ...
    ## If OK then deliver jobs
    for(id in 1:length(children)) {
        ## for each child
        ## initialize message buffer for sending
        .PVM.initsend()
```

```
        ## Divide the work evenly (simple-minded)
        range <- c(start,
                   ifelse((start+chunk-1) > end,
                   end,start+chunk-1))
        ## Take a submatrix
        work <-
            X[(range[1]):(range[2]),,drop=FALSE]
        start <- start + chunk
        ## Pack function name as a string
        .PVM.pkstr(deparse(substitute(FUN)))
        ## Id identifies the order of the job
        .PVM.pkint(id)
        ## Pack submatrix
        .PVM.pkdblmat(work)
        ## Send work
        .PVM.send(children[id], WORKTAG)
    }
    ## Receive any outstanding result
    ## (vector of doubles) from each child
    partial.results <- list()
    for(child in children) {
       ## Get message of type result from any
       ## child.
       .PVM.recv(-1, RESULTAG)
       order  <- .PVM.upkint()
       ## unpack result and restore the order
       partial.results[[order]] <-
           .PVM.upkdblvec()
    }
    ##  unregister from pvm
    .PVM.exit()
    return(unlist(partial.results))
}
```

The corresponding slave script 'slapply.R' is

```
WORKTAG <- 22;  RESULTAG <- 33
## Get parent task id and register
myparent <- .PVM.parent()
## Receive work from parent (a matrix)
buf <- .PVM.recv(myparent, WORKTAG)
## Get function to apply
func <- .PVM.upkstr()
## Unpack data (order, partial.work)
order <- .PVM.upkint()
partial.work <- .PVM.upkdblmat()
## actual computation, using apply
partial.result <- apply(partial.work,1,func)
## initialize send buffer
.PVM.initsend()
## pack order and partial.result
.PVM.pkint(order)
.PVM.pkdblvec(partial.result)
## send it back
.PVM.send(myparent, RESULTAG)
## unregister and exit from PVM
.PVM.exit()
```

An even division of jobs may be far from an optimal strategy, which depends on the problem and in this case, on the network architecture. For example, if some nodes in the cluster are significantly faster than others, one may want send more work to them, but this might be counterbalanced by network distance. Computational overhead (more computation in dividing jobs, network activity due to message sending, etc.) must be considered to achieve better work balance.

Discussion

For parallel Monte Carlo, we need reliable parallel random number generators. The requirements of reproducibility, and hence validation of quality, is important. It isn't clear that selecting different choices of starting seeds for each node will guarantee good randomness properties. The Scalable Parallel Random Number Generators (SPRNG, http://sprng.cs.fsu.edu/) library is one possible candidate. We are working toward incorporating SPRNG into **rpvm** by providing some wrapper functions as well as utilizing existing R functions to generate random numbers from different distributions.

Another challenging problem is to pass higher level R objects through PVM. Because internal data formats may vary across different hosts in the network, simply sending in binary form may not work. Conversion to characters (serialization) appears to be the best solution but there is non-trivial overhead for packing and then sending complicated and/or large objects. This is a similar to the problem of reading in data from files and determining proper data types.

Another future issue is to deploy **rpvm** on Microsoft Windows workstations. Both PVM and R are available under Microsoft Windows, and this is one solution for using additional compute cycles in academic environments.

Bibliography

R. Buyya, editor. *High performance cluster computing: programming and applications, Volume 2*. Prentice Hall, New Jersey, 1999. 6

A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM: Parallel Virtual Machine. A user's guide and tutorial for networked parallel computing*. MIT Press, Massachusetts, 1994. 4

A. Geist, J. A. Kohl, and P. M. Papadopoulos. PVM and MPI: A comparison of features. *Calculateurs Paralleles*, 8, 1996. 4

Michael Na Li
University of Washington
lina@u.washington.edu

Anthony J. Rossini
University of Washington
rossini@u.washington.edu

strucchange: Testing for Structural Change in Linear Regression Relationships

by Achim Zeileis

Introduction

The problem of detecting structural changes arises most often when analyzing time series data with linear regression models, especially in econometrics. Consider the standard linear regression model

$$y_i = x_i^\top \beta_i + u_i \qquad (i = 1, \dots, n),$$

where at time i, y_i is the observation of the dependent variable, x_i is a vector of regressors, β_i is the k-dimensional vector of regression coefficients and u_i is an iid error term. Tests on structural change are concerned with testing the null hypothesis of "no structural change"

$$H_0: \quad \beta_i = \beta_0 \qquad (i = 1, \dots, n),$$

i.e., that the regression coefficients remain constant, against the alternative that the coefficient vector varies over time.

These tests can be divided in two classes, which are differently suitable for certain patterns of deviation from the null hypothesis. The first class are the tests from the generalized fluctuation test framework (Kuan and Hornik, 1995) that can detect various types of structural changes. The second class are the tests from the F test framework (Hansen, 1992; Andrews, 1993), which assume that there is one (unknown) breakpoint under the alternative.

In this article we describe the ideas and methods that have been implemented in the package and that reflect the common features of both classes of tests: how the model for a test can be fitted, the results plotted and finally how the significance can be assessed. First we will introduce the tests and then offer an application on some anthropological data.

Generalized fluctuation tests

Fluctuation tests are either based on estimates or on residuals. The idea of the estimates-based tests is, that if there is a structural change in the data the estimate of the regression coefficients on the basis of all data should be substantially different from the estimates on subsamples of the data that do not contain the structural change(s). But these estimates should be rather similar if the true coefficients remain constant over time. Therefore in this case an empirical process can be computed by the differences of these subsample estimates with the overall estimate. The subsamples are either chosen recursively, i.e., starting with the first k observations and including step by step the next observation, or by a window of constant width that "moves" over the whole sample period. The resulting processes should not fluctuate (deviate from zero) too much under the null hypothesis and—as the asymptotic distributions of these processes are well-known—boundaries can be computed, which are only crossed with a certain controlled probability α. If, on the other hand, the empirical process shows large fluctuation and crosses the boundary, there is evidence that the data contains a structural change. In this case the recursive estimates process should have a peak around the change point, whereas the moving estimates (ME) path will have a strong shift.

Similarly fluctuation processes can be computed based on cumulative or moving sums of two types of residuals: the usual OLS residuals or recursive residuals, which are (standardized) one-step ahead prediction errors. The test based on the CUmulative SUM of recursive residuals (the CUSUM test) was first introduced by Brown et al. (1975) and if there is just one structural break in the coefficients the path will start to leave its zero mean around the break point, because the one-step ahead prediction errors will be large. The OLS-based CUSUM and MOSUM (MOving SUM) test have similar properties as the corresponding estimates-based processes and under a single shift alternative the OLS-CUSUM path should have a peak and the OLS-MOSUM path a shift around the change point. **strucchange** offers a unified approach to deal with these processes: given a formula, which specifies a linear regression model, `efp()` computes an empirical fluctuation process of specified type and returns an object of class `"efp"`. The `plot()` method for these objects plots the process path (and preserves the time series properties if the original data was an object of class `"ts"`) by default together with the corresponding boundaries of level $\alpha = 0.05$. The boundaries alone can also be computed by `boundary()`. Finally a significance test, which also returns a p value, can be carried out using the function `sctest()` (structural change test). The proper usage of these functions will be illustrated in the applications section.

F tests

As mentioned in the introduction, F tests are designed to test against a single shift alternative of the form

$$\beta_i = \begin{cases} \beta_A & (1 \le i \le i_0) \\ \beta_B & (i_0 < i \le n) \end{cases},$$

where i_0 is some change point in the interval $(k, n-k)$. Chow (1960) was the first to suggest a test if the (potential) change point i_0 is known. In his test procedure two OLS models are fitted: one for the observations before and one for those after i_0 and the resulting residuals $\hat{e} = (\hat{u}_A, \hat{u}_B)^\top$ can then be compared with an F test statistic to the residuals $\hat{u}$ from the usual OLS model where the coefficients are just estimated once:

$$F_{i_0} = \frac{(\hat{u}^\top \hat{u} - \hat{e}^\top \hat{e})/k}{\hat{e}^\top \hat{e}/(n-2k)}.$$

For unknown change points (which is the more realistic case) F statistics can be calculated for an interval of potential change points and their supremum can be used as the test statistic. Such a test rejects the null hypothesis if one of the computed F statistics gets larger than a certain critical value or, in other words, if the path of F statistics crosses a constant boundary (defined by the same critical value). The latter shows the possibility to treat sequences of F statistics in a similar way as empirical fluctuation processes: given a formula, which defines a linear regression model, the function `Fstats()` computes a sequence of F statistics for every potential change point in a specified data window and returns an object of class `"Fstats"` (which again preserves the time series properties if the original data had any). Like for `efp` objects there is a `plot()` method available, which plots these F statistics together with their boundary at level $\alpha = 0.05$ or the boundary alone can be extracted by `boundary()`. If applied to `Fstats` objects, `sctest()` computes by default the sup$F$ test statistic and its p value. But there are also two other test statistics available: namely the average of the given F statistics or the exp$F$-functional, which have certain optimality properties (Andrews and Ploberger, 1994).

Application

To demonstrate the functionality of **strucchange** (and to show that there are also applications outside the field of econometrics) we analyze two time series of the number of baptisms (which is almost equivalent to the number of births) and deaths per month in the rural Austrian village Getzersdorf. The data is from the years 1693-1849 (baptisms) and 1710-1841 (deaths) respectively and was collected by the project group "environmental history" from the Institute of Anthropology, Vienna University. The trend of the two time series (extracted by `stl()`) can be seen in Figure 1.

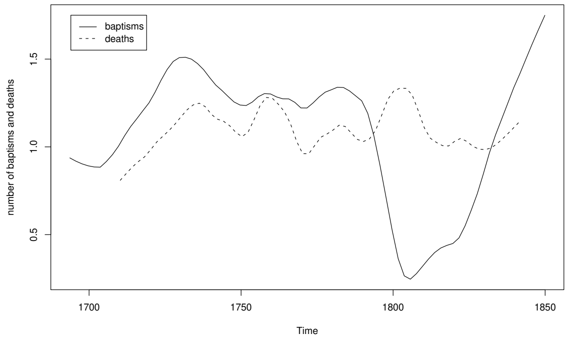

Figure 1: Trend of baptisms and deaths time series from Getzersdorf

We consider the hypothesis that the number of baptisms/deaths remains constant over the sample period. This (almost) implies that the corresponding rate remains constant, because the number of inhabitants remained (almost) constant during the sample period (but is not known explicitely for every month).

The graphs suggest that there was some kind of structural change around 1800 as there is a slight increase in the number of deaths and a dramatic decrease in the number of baptisms. At that time Austria fought against France and Napoleon which explains the decrease of baptisms because the young men were away from home (possibly for several years) and hence couldn't "produce" any offspring.

Analyzing this data with some of the tests from **strucchange** leads to the following results: firstly a Recursive (or Standard) CUSUM model containing just a constant term is fitted to the `ts` objects `baptisms` and `deaths`. The graphical output can be seen in Figure 2.

```
R> baptisms.cus <- efp(baptisms ~ 1,
                       type = "Rec-CUSUM")
R> deaths.cus <- efp(deaths ~ 1,
                     type = "Rec-CUSUM")
R> plot(baptisms.cus); plot(deaths.cus)
```

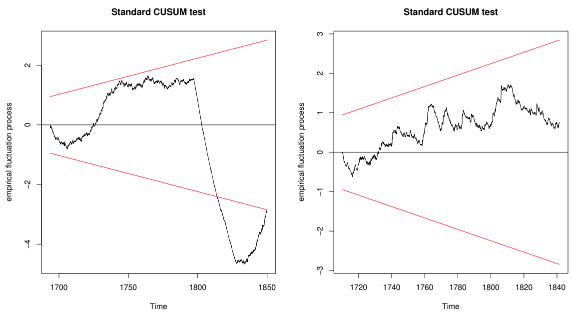

Figure 2: Recursive CUSUM process for baptisms (left) and deaths (right) in Getzersdorf

It can be seen clearly that, whereas the empirical fluctuation process for the death series shows no unusual behaviour, the CUSUM path for the baptisms starts to deviate from its zero mean around 1800, which indicates a structural change at that time. Furthermore there is some deviation from zero at about 1730 (but which is not significant at the 5% level) which corresponds to the increase in baptisms in the original series. Supplementing this graphical analysis a formal significance test can be carried out and a p value can be computed:

```
R> sctest(baptisms.cus); sctest(deaths.cus)

        Standard CUSUM test

data:  baptisms.cus
S = 1.7084, p-value = 1.657e-05

        Standard CUSUM test

data:  deaths.cus
S = 0.6853, p-value = 0.2697
```

Fitting OLS-MOSUM processes leads to very similar results as Figure 3 shows.

```
R> baptisms.mos <- efp(baptisms ~ 1,
                       type = "OLS-MOSUM")
R> deaths.mos <- efp(deaths ~ 1,
                     type = "OLS-MOSUM")
R> plot(baptisms.mos); plot(deaths.mos)
```

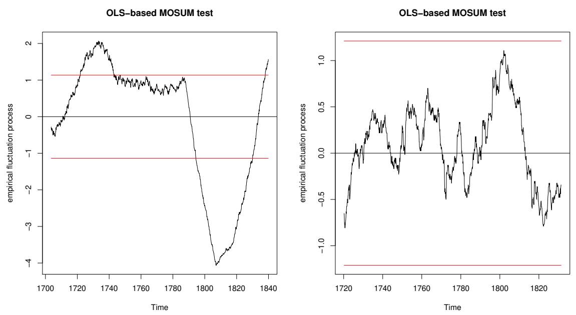

Figure 3: OLS-based MOSUM process for baptisms (left) and deaths (right) in Getzersdorf

The fluctuation of the deaths process remains within its boundaries, although there is a non-significant shift at about 1800. The MOSUM path for the baptisms on the other hand has two shifts: a smaller one around 1730 and a stronger one at 1800, which emphasizes the Recursive CUSUM results.

Finally F statistics are computed for the given times series and the results can be seen in Figure 4.

```
R> baptisms.Fstats <- Fstats(baptisms ~ 1)
R> deaths.Fstats <- Fstats(deaths ~ 1)
R> plot(baptisms.Fstats); plot(deaths.Fstats)
```

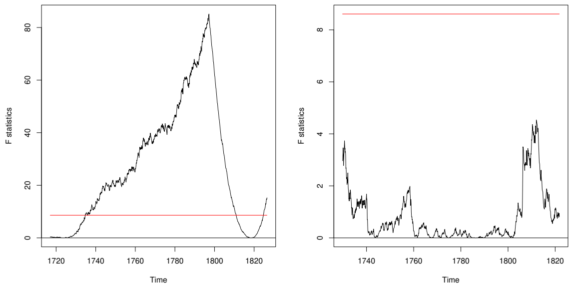

Figure 4: F statistics for baptisms (left) and deaths (right) in Getzersdorf

As in the generalized fluctuation tests no significant result can be achieved for the deaths series, although there is a small peak around 1810, whereas there is an overwhelmingly significant peak at around 1800 for the baptisms. Note that the F statistics just detect the stronger shift at 1800, because they were designed for single shift alternatives.

Summary

strucchange offers a unified framework for generalized fluctuation and F tests for structural change and it extends common significance tests by means to visualize the data and to identify structural changes. More detailed information about the features of **strucchange** can be found in Zeileis et al. (2001).

Bibliography

D. W. K. Andrews. Tests for parameter instability and structural change with unknown change point. *Econometrica*, 61:821–856, 1993. 8

D. W. K. Andrews and W. Ploberger. Optimal tests when a nuisance parameter is present only under the alternative. *Econometrica*, 62:1383–1414, 1994. 9

R. L. Brown, J. Durbin, and J. M. Evans. Techniques for testing the constancy of regression relationships over time. *Journal of the Royal Statistical Society*, B 37:149–163, 1975. 8

G. C. Chow. Tests of equality between sets of coefficients in two linear regressions. *Econometrica*, 28: 591–605, 1960. 9

B. E. Hansen. Tests for parameter instability in regressions with I(1) processes. *Journal of Business & Economic Statistics*, 10:321–335, 1992. 8

C.-M. Kuan and K. Hornik. The generalized fluctuation test: A unifying view. *Econometric Reviews*, 14: 135–161, 1995. 8

A. Zeileis, F. Leisch, K. Hornik, and C. Kleiber. strucchange: An R package for testing for structural change in linear regression models. Report 55, SFB "Adaptive Information Systems and Modelling in Economics and Management Science", May 2001. URL http://www.wu-wien.ac.at/am/reports.htm#55. 10

Achim Zeileis
Technische Universität Wien, Austria
zeileis@ci.tuwien.ac.at

Programmer's Niche: Macros in R

Overcoming R's virtues

by Thomas Lumley

A familiar source of questions on the R mailing lists is the newly converted R user who is trying to write SAS or Stata code in R. Bill Venables then points out to them that R is not a macro language, and gently explains that there is a much easier solution to their problems. In this article I will explain what a macro is, why it's good that R isn't a macro language, and how to make it into one.

There are two reasons for this. It has been famously observed[1] that a Real Programmer can write Fortran code in any language, and it is similarly an interesting exercise to see how R can implement macros. Secondly, there are a few tasks for which macros are genuinely useful, which is why languages like LISP, for example, provide them.

What is a macro language?

Suppose you have a series of commands

```
table(treatment, gender)
table(treatment, race)
table(treatment, age.group)
table(treatment, hospital)
table(treatment, diabetic)
```

These commands can be created by taking the skeleton

```
table(treatment, variable)
```

substituting different pieces of text for `variable`, and evaluating the result. We could also repeatedly call the `table()` function with two arguments, the first being the values of `treatment` and the second being the values of the other variable.

R takes the latter approach: evaluate the arguments then use the values. We might define

```
rxtable <- function(var){
  table(treatment, var)
}
```

Stata typically takes the former approach, substituting the arguments then evaluating. The 'substitute then evaluate' approach is called a *macro expansion*, as opposed to a *function call*. I will write this in pseudo-R as

```
rxtable <- macro(var){
  table(treatment, var)
}
```

Why not macros?

In this simple example it doesn't make much difference which one you use. In more complicated examples macro expansion tends to be clumsier. One of its advantages is that you get the actual argument names rather than just their values, which is useful for producing attractive labels, but R's lazy evaluation mechanism lets you do this with functions.

One problem with macros is that they don't have their own environments. Consider the macro

```
mulplus <- macro(a, b){
  a <- a+b
  a * b
}
```

to compute $(a+b)(b)$. This would work as a function, but as a macro would have undesirable side-effects: the assignment is not to a local copy of `a` but to the original variable. A call like `y <- mulplus(x, 2)` expands to `y <- {x<-x+2; x*2}`. This sets `y` to the correct value, $2x+4$, but also increments `x` by 2. Even worse is `mulplus(2, x)`, which tries to change the value of 2, giving an error.

We could also try

```
mulplus <- macro(a, b){
 temp <- a+b
 temp * b
}
```

This appears to work, until it is used when we already have a variable called `temp`. Good macro languages need some way to provide variables like `temp` that are guaranteed not to already exist, but even this requires the programmer to declare explicitly which variables are local and which are global.

The fact that a macro naturally tends to modify its arguments leads to one of the potential uses of macro expansion in R. Suppose we have a data frame

[1] "Real Programmers don't use Pascal" by Ed Post — try any web search engine

in which one variable is coded -9 for missing. We need to replace this with NA, eg,

```
library(survival)
data(pbc)
pbc$bili[pbc$bili %in% -9] <- NA
```

For multiple missing values and many variables this can be tedious and error-prone. Writing a function to do this replacement is tricky, as the modifications will then be done to a copy of the data frame. We could use the <<- operator to do the assignment in the calling environment. We then face the problem that the function needs to know the names pbc and bili. These problems are all soluble, but indicate that we may be going about things the wrong way.

We really want to take the expression

```
df$var[df$var %in% values] <- NA
```

and substitute new terms for df, var and values, and then evaluate. This can be done with the substitute() function

```
eval(substitute(
  df$var[df$var %in% values] <- NA,
  list(df=quote(pbc), var=quote(bili),
       values=-9)))
```

but this is even more cumbersome than writing out each statement by hand. If we could define a macro

```
setNA<-macro(df, var, values){
  df$var[df$var %in% values] <- NA
}
```

we could simply write

```
setNA(pbc, bili, -9)
```

Using macro expansion in R

The example using substitute() shows that macro expansion is possible in R. To be useful it needs to be automated and simplified. Adding macro to the language as a new keyword would be too much work for the benefits realised, so we can't quite implement the notation for macros that I have used above. We can keep almost the same syntax by defining a function defmacro() that has the argument list and the body of the macro as arguments.

Using this function the setNA macro is defined as

```
setNA <- defmacro(df, var, values, expr={
  df$var[df$var %in% values] <- NA
})
```

and used with

```
setNA(pbc, bili, -9).
```

The argument list in defmacro can include default arguments. If -9 were a commonly used missing value indicator we could use

```
setNA <- defmacro(df, var, values = -9, expr={
  df$var[df$var %in% values] <- NA
})
```

Macros can also provide another implementation of the 'density of order statistics' example from the R-FAQ. The density of the rth order statistic from a sample of size n with cdf F and density f is

$$f_{(r),n}(x) = \frac{n(n-1)!}{(n-r)!(r-1)!}F(x)^{r-1}(1-F(x))^{n-r}f(x).$$

The FAQ explains how to use lexical scope to implement this, and how to use substitute() directly. We can also use a macro

```
dorder <- defmacro(n, r, pfun, dfun,expr={
  function(x) {
    con <- n*choose(n-1, r-1)
    con*pfun(x)^(r-1)*(1-pfun(x))^(n-r)*dfun(x)
  }
})
```

so that the median of a sample of size 11 from an exponential distribution has density

```
dmedian11 <- dorder(11, 6, pexp, dexp)
```

In this case lexical scope may be an easier solution, but 'functions to write functions' are a standard use of macros in LISP.

So how does it work?

The function defmacro() looks like

```
defmacro <- function(..., expr){
  expr <- substitute(expr)
  a <- substitute(list(...))[-1]
  ## process the argument list
  nn <- names(a)
  if (is.null(nn)) nn <- rep("", length(a))
  for(i in seq(length=length(a))) {
    if (nn[i] == "") {
      nn[i] <- paste(a[[i]])
      msg <- paste(a[[i]], "not supplied")
      a[[i]] <- substitute(stop(foo),
                    list(foo = msg))
    }
  }
  names(a) <- nn
  a <- as.list(a)
  ## this is where the work is done
  ff <- eval(substitute(
          function(){
            tmp <- substitute(body)
            eval(tmp, parent.frame())
          },
          list(body = expr)))
  ## add the argument list
  formals(ff) <- a
  ## create a fake source attribute
  mm <- match.call()
  mm$expr <- NULL
  mm[[1]] <- as.name("macro")
```

```
  attr(ff, "source") <- c(deparse(mm),
                          deparse(expr))
  ## return the 'macro'
  ff
}
```

The kernel of `defmacro()` is the call

```
ff <- eval(substitute(
    function(){
      tmp <- substitute(body)
      eval(tmp, parent.frame())
    },
    list(body = expr)))
```

In the `setNA` example this creates a function

```
function(){
  tmp <- substitute(
        df$var[df$var %in% values] <- NA)
  eval(tmp, parent.frame())
}
```

that performs the macro expansion and then evaluates the expanded expression in the calling environment. At this point the function has no formal argument list and most of `defmacro()` is devoted to creating the correct formal argument list.

Finally, as printing of functions in R actually uses the `source` attribute rather deparsing the function, we can make this print in a more user-friendly way. The last lines of `defmacro()` tell the function that its source code should be displayed as

```
macro(df, var, values){
  df$var[df$var %in% values] <- NA
}
```

To see the real source code, strip off the `source` attribute:

```
attr(setNA, "source") <- NULL
```

It is interesting to note that because `substitute` works on the parsed expression, not on a text string, `defmacro` avoids some of the problems with C preprocessor macros. In

```
mul <- defmacro(a, b, expr={a*b})
```

a C programmer might expect `mul(i, j + k)` to expand (incorrectly) to `i*j + k`. In fact it expands correctly, to the equivalent of `i*(j + k)`.

Conclusion

While `defmacro()` has many (ok, one or two) practical uses, its main purpose is to show off the powers of `substitute()`. Manipulating expressions directly with `substitute()` can often let you avoid messing around with pasting and parsing strings, assigning into strange places with `<<-` or using other functions too evil to mention. To make `defmacro` really useful would require local macro variables. Adding these is left as a challenge for the interested reader.

Thomas Lumley
University of Washington, Seattle
tlumley@u.washington.edu

More on Spatial Data Analysis

by Roger Bivand

Introduction

The second issue of *R News* contained presentations of two packages within spatial statistics and an overview of the area; yet another article used a fisheries example with spatial data. The issue also showed that there is still plenty to do before spatial data is as well accommodated as date-time classes are now. This note will add an introduction to the **splancs** package for analysing point patterns, mention briefly work on packages for spatial autocorrelation, and touch on some of the issues raised in handling spatial data when interfacing with geographical information systems (GIS).

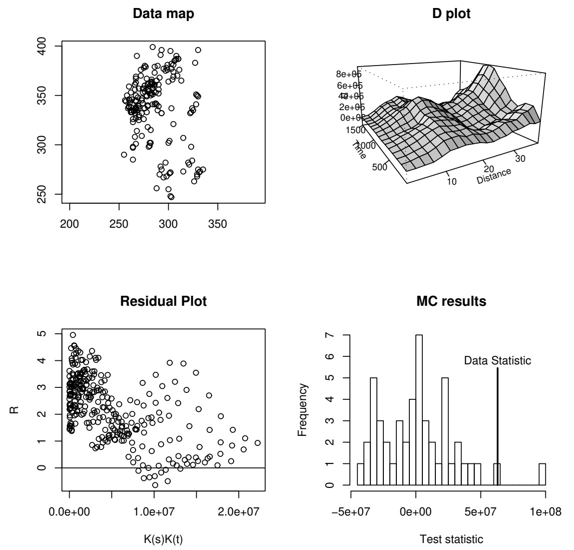

Figure 1: Burkitt's lymphoma — `stdiagn()` output.

The splancs package

The **splancs** package in R is based on a package of that name written in S and FORTRAN in the early 1990's by Barry Rowlingson and Peter Diggle to provide a tool for display and analysis of spatial point pattern data. The functions provided by the package are described in detail in Rowlingson and Diggle (1993), and additional space-time and raised incidence functions introduced in version 2 are described in file 'Sp2doc.ps', available from Barry Rowlingson's web pages[1]. Some of the functionality duplicates that already in the **spatial** package (Venables and Ripley, 1999, Chapter 14) in the **VR** bundle, a recommended package, see Ripley (1981) and Diggle (1983). It is worth noting that the **splancs** functions use an arbitrary polygon to define the study region in the computation of edge effects. The name of the package perhaps plays on the continuing strength of Lancaster University in spatial statistics.

The examples and exercises in Bailey and Gatrell's 1995 spatial data analysis textbook Bailey and Gatrell (1995) add a lot to teaching from it. They are part of software called INFO-MAP packaged with the book and running under DOS. Replicating the functionality needed to study the point pattern examples under R has been important in porting **splancs** to R, especially as the book actually reproduces output from **splancs**. Consequently, the topics covered best by the port are those that carry most weight in Bailey and Gatrell: kernel estimation, nearest neighbour distances, the K function, tests for nearest neighbours and the K function based on complete spatial randomness, and thanks to a contribution by Giovanni Petris, also on the Poisson cluster process. These functions cover Chapter 3; with permission from the authors, the package includes data sets taken from the book.

Topics treated in Chapter 4 include space-time clustering, correcting for variations in the population at risk, and testing for clustering around a specific point source. The **splancs** functions for the first two groups are now fully documented and have datasets that allow the user to re-create data visualizations as shown in print; these are usually the source of the graphics for `example()` for the function concerned. Running `example(stdiagn)` generates the output shown in Figure 1, corresponding to graphics on page 125 in Bailey and Gatrell — here to examine space-time clustering in cases of Burkitt's lymphoma in the West Nile District of Uganda, 1961–75.

Because porting **splancs** has been influenced by the textbook used for teaching, some parts of the original material have been omitted — in particular the `uk()` function to draw a map of England, Scotland and Wales (now partly available in package **blighty**). An alternative point in polygon algorithm has been added for cases where the result should not be arbitrary for points on the polygon boundary (thanks to Barry Rowlingson and Rainer Hurling). In conclusion, learning about point pattern analysis ought not to start by trying out software without access to the underlying references or textbooks, because of the large number of disparate methods available, and the relatively small volume of analyses conducted using them.

[1] http://www.maths.lancs.ac.uk/~rowlings/Splancs/

Spatial autocorrelation

As pointed out in Brian Ripley's overview in the previous issue, the availability of the commercial **S+SpatialStats** module for S-PLUS does make the duplication of implementations less interesting than trying out newer ideas. A topic of some obscurity is that of areal or lattice data, for which the available data are usually aggregations within often arbitrary tessellations or zones, like counties. They differ from continuous data where attributes may be observed at any location, as in geostatistics, often because the attributes are aggregates, like counts of votes cast in electoral districts. While this is perhaps not of mainstream interest, there is active research going on, for example Bavaud (1998), Tiefelsdorf et al. (1999) and Tiefelsdorf (2000).

This activity provides some reason to start coding functions for spatial weights, used to describe the relationships between the spatial zones, and then to go on to implement measures of spatial autocorrelation. So far, a package **spweights** has been released to handle the construction of weights matrices of various kinds, together with associated functions. Comments and suggestions from Elena Moltchanova, Nicholas Lewin-Koh and Stephane Dray have helped, allowing lists of neighbours to be created for distance thresholds and bands between zone centroids, by triangulation, by graph defined neighbourhoods, by k-nearest neighbours, by finding shared polygon boundaries, and reading legacy format sparse representations. Two classes have been established, `nb` for a list of vectors of neighbour indices, and `listw` for a list with an `nb` member and a corresponding list of weights for the chosen weighting scheme. Such a sparse representation permits the handling of $n \times n$ weights matrices when n is large.

An active area of research is the construction of of tessellations of the plane from points, or other objects in a minimum amount of time, since without limiting the search area computations can quickly exceed $O(n^2)$. Research in this area is active in computational geometry, machine learning, pattern recognition, operations research and geography. Reduction of computation in these searching operations requires data structures that facilitate fast range searching and query. This is still an area where R is deficient in relation to packages like S-PLUS and Matlab which both support quadtrees for accelerated neighbourhood searching.

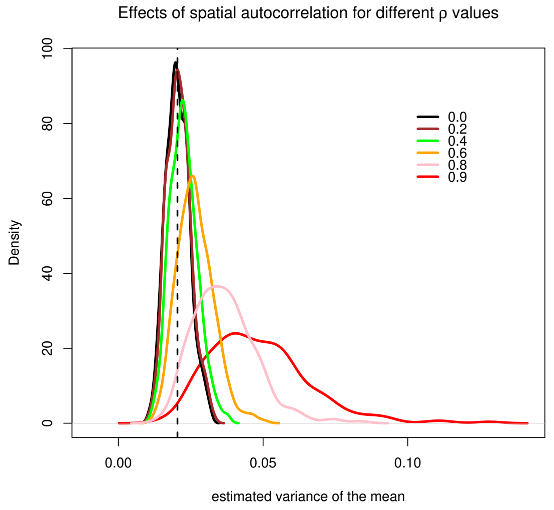

Figure 2: Simulation of the effects of simultaneous autocorrelation on estimates of the variance of the mean $\sigma^2_{\bar{x}}$

Autocorrelation in a sample is important to consider, since the presence of autocorrelation can severely bias the estimation of the variance of the sample moments. Figure 2 is the output of `example(invIrM)`, illustrating the effect of increasing the simultaneous autocorrelation parameter ρ on estimates of the variance $\sigma^2_{\bar{x}}$ of the mean. The simulation used 500 samples of ϵ, a random normal variate with zero mean and unit variance on a 7×7 lattice on a torus (a square grid mapped onto a circular tube to remove edge effects). Autocorrelation is introduced into x by $\mathbf{x} = \sigma(\mathbf{I} - \rho\mathbf{W})^{-1}\boldsymbol{\epsilon}$, where $w_{ij} > 0$ when i, j are neighbours, under certain conditions on ρ (Cliff and Ord, 1981, p. 152). (Sparse representations are not used because the inverse matrix is dense.) The vertical line indicates the estimator assuming independence of observations, for known $\sigma^2 = 1$. Since strong positive autocorrelation erodes the effective number of degrees of freedom markedly, assuming independence with spatial data may be brave.

Tests for autocorrelation using these matrices are implemented in **sptests** — so far Moran's I, Geary's C and for factors, the same-colour join count test. Using the `USArrests` and `state` data sets, and dropping Alaska and Hawaii, we can examine estimates of Moran's I:

$$I = \frac{n}{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}} \frac{\sum_{i=1}^{n}\sum_{j=1}^{n} w_{ij}(x_i - \bar{x})(x_j - \bar{x})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

for weights matrices using row-standardized k-nearest neighbours schemes for $k = 1, \ldots, 5$, for $w_{ij} = 1/k$ (Cliff and Ord, 1981, p. 17). Moran's I for assault arrests (per 100,000) for 48 US continental states in 1975 for increasing k is shown in the following table: the expectation of I is known $(-1/(n-1))$ and the variance is calculated under randomisation. 'Rank' is the rank of the observed statistic when added to the values from 499 permutations.

k	Moran's I	Variance	Std. deviate	Rank
1	0.405	0.0264	2.63	497
2	0.428	0.0161	3.53	500
3	0.306	0.0109	3.14	500
4	0.294	0.0083	3.46	498
5	0.282	0.0066	3.74	500

As can be seen, it does seem likely that observed rates of assault arrests of k-nearest neighbour states are positively autocorrelated with each other. Using these packages, this may be run for $k = 4$ by:

```
Centers48 <-
  subset(data.frame(x=state.center$x,
                    y=state.center$y),
         !state.name %in% c("Alaska", "Hawaii"))
Arrests48 <-
  subset(USArrests, !rownames(USArrests) %in%
    c("Alaska", "Hawaii"))
k4.48 <- knn2nb(knearneigh(as.matrix(Centers48),
  k=4))
moran.test(x=Arrests48$Assault,
  listw=nb2listw(k4.48))
moran.mc(x=Arrests48$Assault,
  listw=nb2listw(k4.48), nsim=499)
```

where `knearneigh`, `knn2nb` and `nb2listw` are in **spweights** and `moran.test` and `moran.mc` in **sptests**. The exact distribution of Moran's I has been solved as a ratio of quadratic forms Tiefelsdorf and Boots (1995) but is not yet implemented in the package. The MC solution is however more general since it can be applied to any instance of the General Cross Product statistic Hubert et al. (1981).

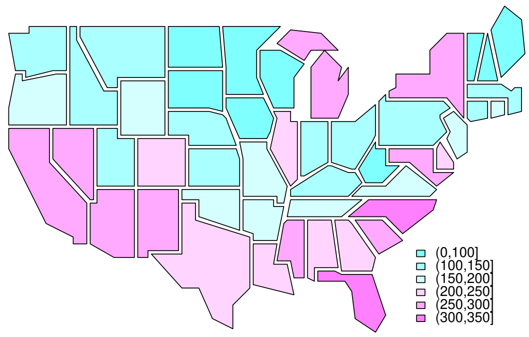

Figure 3: Assault arrests (per 100,000) for 48 US continental states in 1975.

Spatial locations

The `state.center` data used in the above example are documented as being in geographical coordinates, eastings and northings measured in degrees. In finding near neighbours, distances were measured as if the points representing the states' location were on the plane, which they are not. Metadata about the projection and measurement units of spatial data are of importance in the same way that adequate handling of date and time objects may matter. There may be systematic regularities within the data series itself that are obscured by a lack of registration, and such a lack may make it impossible to combine the data at hand with other data with positional accuracy. In this case, the point locations may be projected onto a plane, here using the Universal Transverse Mercator projection, a standard ellipsoid and datum, for zone 15 (centring the plane East-West on Iowa) and measuring in km, using PROJ.4 software[2].

```
write.table(Centers48, file="ll48.txt",
  row.names=FALSE, col.names=FALSE)
system(paste("proj -m 1:1000 +proj=utm",
  "+zone=15 ll48.txt > utm48.txt"))
Centers48.utm15a <- read.table("utm48.txt")
k3.48utm15 <- knn2nb(knearneigh(as.matrix(
  Centers48.utm15), k=3))
summary(diffnb(k3.48utm15, k3.48, verbose=FALSE))
```

Comparing the neighbour lists for $k = 3$ nearest neighbours for the two sets of coordinates shows that, of the 48 states, 29 had the same 3 nearest neighbours, 18 changed one nearest neighbour, and Idaho changed 2. Despite this, the results of testing for spatial autocorrelation were unchanged, confirming the strong impression of spatial structure visualized in Figure 3.

k	Moran's I	Variance	Std. deviate	Rank
2	0.484	0.0164	3.94	500
3	0.327	0.0108	3.36	499
4	0.315	0.0084	3.67	499

While in this case there is no change in the conclusion drawn, it seems to a geographer to be worth being as careful with spatial metadata as we now can be with temporal metadata. One approach implemented in the **GRASS** package for interfacing R with the GPL'ed geographical information system GRASS[3] is to store the metadata in a separate object recording the current settings of the GIS: region of interest, projection, measurement units, and raster cell resolution. This secures the use of the same metadata on both sides of the interface for a given work session, but separate data objects, such as sets of point coordinates, do not have their own embedded metadata. An alternative approach is used in the new package for importing and exporting portable anymap graphics files (**pixmap**). Here metadata are attached to data objects through attribute values, as `ts` does with time series objects.

In the same way that it has taken time for dates and times to find forms that are both powerful and sufficiently general, spatial data will find a class structure probably with metadata attributes—even date/time metadata attributes. It is obvious that duplicating GIS functionality in R is not a good solution, but much spatial data analysis needs a blend of positional registration, visualization and analytical tools that are not available within the GIS either. This means that the GIS-style metadata need to accompany the data from the GIS to the analysis environment and back again. Connections functions now allow us to move data very freely, but having to rely on intervention by the analyst to make sure that metadata follows is not a good solution.

[2] http://www.remotesensing.org/proj/
[3] http://grass.itc.it/

Prospects

There is now a range of packages for spatial statistics in R. They all have different object structures for positional data, and metadata is handled differently. R still does not have a `map` function on CRAN, but sorting out how to interface with spatial data should help with this. More efficient mechanisms for exchanging data with GIS will add both to access to modern statistical tools by GIS users, and to more appropriate treatment of spatial metadata in spatial statistics. Happily, GPL'ed software like that used for projection above is under active development, and standards for spatial data and spatial reference systems are gelling. These can be given R package wrappers, but there is, though, plenty to do!

Bibliography

T. C. Bailey and A. C. Gatrell. *Interactive spatial data analysis*. Longman, Harlow, 1995. 14

F. Bavaud. Models for spatial weights: a systematic look. *Geographical Analysis*, 30:152–171, 1998. 14

A. D. Cliff and J. K. Ord. *Spatial processes — models and applications*. Pion, London, 1981. 15

P. J. Diggle. *Statistical analysis of spatial point patterns*. Academic Press, London, 1983. 14

L. J. Hubert, R. G. Golledge, and C. M. Costanzo. Generalized procedures for evaluating spatial autocorrelation. *Geographical Analysis*, 13:224–233, 1981. 15

B. D. Ripley. *Spatial statistics*. Wiley, New York, 1981. 14

B. Rowlingson and P. J. Diggle. Splancs: spatial point pattern analysis code in S-PLUS. *Computers and Geosciences*, 19:627–655, 1993. 14

M. Tiefelsdorf. *Modelling spatial processes*, volume 87 of *Lecture notes in earth sciences*. Springer, Berlin, 2000. 14

M. Tiefelsdorf and B. Boots. The exact distribution of Moran's I. *Environment and Planning A*, 27:985–999, 1995. 15

M. Tiefelsdorf, D. A. Griffith, and B. Boots. A variance-stabilizing coding scheme for spatial link matrices. *Environment and Planning A*, 31:165–180, 1999. 14

W. N. Venables and B. D. Ripley. *Modern applied statistics with S-PLUS*. Springer, New York, 1999. 14

Roger Bivand
Economic Geography Section, Department of Economics, Norwegian School of Economics and Business Administration, Bergen, Norway
Roger.Bivand@nhh.no

Object-Oriented Programming in R

by John M. Chambers & Duncan Temple Lang

Although the term *object-oriented programming* (OOP) is sometimes loosely applied to the use of methods in the S language, for the computing community it usually means something quite different, the style of programming associated with Java, C++, and similar languages. OOP in that sense uses a different basic computing model from that in R, specifically supporting mutable objects or references. Special applications in R can benefit from it, in particular for inter-system interfaces to OOP-based languages and event handling. The **OOP** module in the Omegahat software implements the OOP model for computing in R.

S language philosophy and style

When you write software in R, the computations are a mixture of calls to functions and assignments. Although programmers aren't usually consciously thinking about the underlying philosophy or style, there is one, and it affects how we use the language.

One important part of the S language philosophy is that functions ordinarily don't have side effects on objects. A function does some computation, perhaps displays some results, and returns a value. Nothing in the environment from which the function was called will have been changed behind the scenes.

This contrasts with languages which have the notion of a pointer or *reference* to an object. Passing a reference to an object as an argument to a function or routine in the language allows the called function to alter the object referred to, in essentially arbitrary ways. When the function call is complete, any changes to that object persist and are visible to the caller.

In general, S functions don't deal with references, but with objects, and function calls return objects, rather than modifying them. However, the language does include assignment operations as an explicit means of creating and modifying objects in the local frame. Reading the S language source, one can immediately see where any changes in an object can take place: only in the assignment operations for that specific object.[1]

Occasionally, users ask for the addition of references to R. Providing unrestricted references would radically break the style of the language. The "raw pointer" style of programming used in C, for example, would be a bad temptation and could cause chaos for R users, in our opinion.

A more interesting and potentially useful alternative model, however, comes from the languages that support OOP in the usual sense of the term. In these languages, the model for programming is frequently centered around the definition of a class of objects, and of methods defined for that class. The model does support object references, and the methods can alter an object remotely. In this sense, the model is still sharply different from ordinary R programming, and we do not propose it as a replacement.

However, there are a number of applications that can benefit from using the OOP model. One class of examples is inter-system interfaces to languages that use the OOP model, such as Java, Python, and Perl. Being able to mimic in R the class/method structure of OOP software allows us to create a better and more natural interface to that software. R objects built in the OOP style can be used as regular objects in those languages, and any changes made to their state persist. The R code can work directly in terms of the methods in the foreign language, and much of the interface software can be created automatically, using the ability to get back the metadata defining classes (what's called *reflectance* in Java).

Mutable objects (i.e., object references) are also particularly useful when dealing with asynchronous events. For example, when a user clicks on a help button in a graphical user interface (GUI), we might first check to see if we have previously created the help window and if not, create the window and store

[1] Assuming that the function doesn't cheat. Almost anything is possible in the S language, in that the evaluator itself is available in the language. For special needs, such as creating programming tools, cheating this way is admirable; otherwise, it is unwise and strongly deprecated.

a reference to it for use in the future. Here, updating the state of an object associated with the help action is convenient and natural. Similarly, cumulating data from a connection or stream when becomes available can be done easily by updating the state of an OOP object.

The OOP model

In the OOP languages of interest here, functions are no longer the central programming tool. The basic unit of software is the definition of a class of objects. The class definition can include the data structure (the *slots* or *fields*) of the class, and the *methods* that can be invoked on objects from the class.

Methods in this model play somewhat the role of functions in R. But, in contrast to methods in R, these methods are associated with the class and the objects or particular realizations of the class. You invoke methods *on* an object. To illustrate, let's use an example in R. A simple application of OOP-style computing that we will discuss below is to create R objects that represent FTP (File Transfer Protocol) connections to remote sites.

One of the things you need to do with an FTP connection is to login. In the OOP model, `login` is a method defined for this class of objects. One invokes this method on an object. So, if the S object `franz` is an instance from an appropriate FTP class, the computation might look like:

```
franz$login("anonymous", "jmc@lucent.com")
```

In words, this says: for the object `franz` find the appropriate definition of the `login` method, and call it with the two strings as additional arguments. The exact notation depends on the language. We're using the familiar `$` operator, which in fact turns out to be convenient for implementing OOP programming in R. Java, Python, Perl, and other languages each have slightly different notation, but the essential meaning carries over.

Invoking methods rather than calling functions is the main difference in appearance. Object references and the ability to change an object through a method are the main differences in what actually happens. Where an application naturally suits such references, the OOP model often fits well. Object references and OOP suit the example of an FTP connection.

FTP is a simple but effective way to connect to a remote site on the Web and transfer data back and forth. Creating a connection to a particular site from a session in R, say, gives us a "thing"—an object, let's say. Unlike more usual R objects such as vectors of numbers, an FTP connection is very much a single thing, referring to that actual connection to the remote site. Computations may change the state of that object (e.g., whether we have successfully logged in to the site, or where we are currently in the file system). When they do, that changed state needs to be visible to all copies of the object: whatever R function call we're in, the FTP object for this connection *refers to* the same connection object.

In contrast, if one R function passes a vector of numbers to another R function, and that function rearranges its copy of the numbers, it's *not* the usual model that both copies change! We write R software all the time in the confidence that we can pass arguments to other functions without worrying about hidden side effects.

Two different computational models, each useful in the right context.

OOP in R

The ability to have object references, in effect, in R can be implemented fairly directly, through a programming "trick". The closure concept in R allows functions to be created that can read and assign to variables in their parent environment. These variables are then in effect references, which can be altered by the functions, acting like OOP methods. Using closures, we might implement the FTP class in the following manner:

```
FTP <- function(host) {
  con <- NULL

  login <- function(id, passwd) {
    if(!is.null(con)) {
      stop("already logged in")
    }
    con <<- .Call("FTPLogin", machine,
                  id, passwd)
  }

  return(list(login=login))
}
```

We can use this

```
franz <- FTP("franz.stat.wisc.edu")
franz$login("anonymous", "jmc@lucent.com")
```

The first line creates a new instance of the FTP class with its own version of the `machine` and `con` variables. The call to `login()` updates the object's `con` value and subsequent calls can see this new value. More information and examples of closures are given in Gentleman and Ihaka (2000).

This approach is simple and fairly efficient, and can be quite useful. However, we are proposing here a somewhat more formal mechanism. Being more formal is helpful, we think, partly because the mapping to analogous OOP systems in other languages is then clearer. Formal definitions in software also have the advantage that they can be queried to let software help write other software. We use such "reflectance" in other languages when building interfaces from R, and being formal ourselves brings similar advantages. Interfaces *from* other languages can query OOP class definitions in R. For example, we can automatically define Java or Python classes that mimic or even extend R classes. Programming with formal OOP classes in R should be easier also, since the formal approach provides tools for defining classes and methods similar to those that have worked well in other languages, while at the same time being simple to use in R. Finally, the formal OOP approach makes it more feasible to have an OOP formalism that is compatible between R and S-Plus, making software using the approach available to a wider audience.

Defining classes

OOP programming begins by defining a class; specifically by creating a *class object* with a call to the `setOOPClass` function:

```
> setOOPClass("FTP")
```

The call to `setOOPClass` creates an OOP class definition object, with `"FTP"` as the class name, and also assigns the object with the same name. Class objects contain definitions for the methods available in the class. Objects from the class will usually contain data, stored in specified *fields* in the class. In our model, these fields are not accessed directly; access is encapsulated into methods to get and set fields. Classes can inherit from other OOP classes, and the class can itself have methods and fields. The class object, `FTP`, is an OOP object itself, so we can use OOP methods to set information in the class object.

For our FTP example, the class contains two fields to hold the name of the machine and the connection object:

```
> FTP$setFields(machine = "character",
                con = "connection")
```

This has the side effect of creating methods for setting and getting the values of these fields. To use the class, we create a constructor function which is responsible for storing the name of the host machine.

```
FTP$defineClassMethod(
  "new", function(machine) {
    x <- super(new())
    x$setMachine(machine)
    x
  }
)
```

Next we define the `login()` method for objects of this class.

```
FTP$defineMethod(
  "login", function(id, passwd) {
    setConnection(
      .Call("FTPLogin",
            getMachine(), id, passwd))
  }
)
```

The OOP methods `defineMethod` and `defineClassMethod` modify the object `FTP`.

Besides defining classes directly, R programmers can create interfaces to class definitions in other languages. If our `FTP` class used the interface to Perl, it might be created directly from a known class in Perl:

```
> FTP <- PerlClass("FTP", package="Net")
```

Methods for such classes can be defined automatically, using reflectance information. The extent to which this happens varies with the other language—Java provides a lot of information, Perl less.

Once the class is defined, objects can be created from it.

```
> franz <- FTP$new("franz.stat.wisc.edu")
```

Objects from an OOP class can be assigned and passed to functions just like any objects in R. But they are fundamentally different, in that they contain an object reference. If `franz` is passed to an R function, and that function calls an OOP method that changes something in its argument, you can expect to see the effect of the change in the object `franz` as well.

Further information

The software and documentation for the OOP package for R is available from the Omegahat Web site at http://www.omegahat.org/OOP/.

Bibliography

Robert Gentleman and Ross Ihaka. Lexical scope and statistical computing. *Journal of Computational and Graphical Statistics*, 9(3):491–508, September 2000. 18

John Chambers
Bell Labs, Murray Hill, New Jersey, U.S.A
jmc@research.bell-labs.com

Duncan Temple Lang
Bell Labs, Murray Hill, New Jersey, U.S.A
duncan@research.bell-labs.com

In Search of C/C++ & FORTRAN Routines

by Duncan Temple Lang

One of the powerful features of the S language (i.e., R and S-Plus) is that it allows users to dynamically (i.e., in the middle of a session) load and call arbitrary C/C++ and FORTRAN routines. The `.C()`, `.Call()`[1], `.Fortran()` and `.External()` functions allow us to call routines in third-party libraries such as NAG, Atlas, Gtk, while the data is created and managed in S. Importantly, it also allows us to develop algorithms entirely in S and then, if needed, gradually move the computationally intensive parts to more efficient C code. More recently, we have generalized these interfaces to "foreign" languages to provide access to, for example, Java, Python, Perl and JavaScript.

In this article we discuss some of the pitfalls of the current mechanism that R uses to locate these native routines. Then we present a new mechanism which is more portable, and offers several beneficial side effects which can make using native routines more robust and less error-prone.

The current system

The `dyn.load()` function in R loads a C or FORTRAN shared library or dynamically linked library (*DLL*) into the session and makes *all* the routines in that library available to the R user. This allows S users to call any symbol in that library, including variables! The low-level details of `dyn.load()` are usually provided by the user's operating system, and in other cases can be implemented with some clever, non-portable code. While we get much of this for free, there are many subtle but important differences across the different operating systems on which R runs, Windows, Linux, Solaris, Irix, Tru64, Darwin, to name a few. And worse still, the behavior depends on both the machines and the user's own configurations. Therefore, porting working code to other platforms may be non-trivial.

Many uses of `dyn.load()` are quite straightforward, involving C code that doesn't make use of any other libraries (e.g., the **mva** and **eda** packages). Things become more complex when that C code uses other libraries (e.g., the NAG and Lapack libraries), and significantly more variable when those third party libraries in turn depend on other libraries. The main issue for users is how are these other libraries found on the system. Developers have to be careful that symbols in one library do not conflict with those in other libraries and that the wrong symbols do not get called, directly or indirectly. There exists a non-trivial chance of performing computations with the wrong code and getting subtly incorrect results. If one is lucky, such errors lead to catastrophic consequences and not hard to identify errors in the results.

And, of course, regardless of finding the correct routine, users also have to be careful to pass the correct number and type of arguments to the routines they are intending to call. Getting this wrong typically terminates the S session in an inelegant manner. (Read "crash"!)

Generally, while DLLs have many benefits, they can also be quite complicated for the user to manage precisely. Why I am telling you about the potential pitfalls of the dynamic loading facility, especially when for most users things have worked quite well in the past? One reason is that as we use R in more complex settings (e.g., embedded in browsers, communicating with databases) these problems will become more common. Also, the main point, however, is that we only use a small part of the dynamic loading capabilities of the operating system but have to deal with all of the issues. A simpler mechanism is more appropriate for most S users. R 1.3.0 allows developers of R packages and DLLs to use a more coherent and predictable mechanism for making routines available to the `.C()`, `.Call()`, `.Fortran()` and `.External()` functions.

In the next page or two, we'll take a brief look at an example of using this new mechanism and explain how it works. We'll also discuss how we will be able to use it to make accessing native code more resistant to errors and also automate aspects of passing data to C routines from S and back. The new **Slcc** package has potential to programmatically generate S and C code that provides access to arbitrary C libraries.

The default mechanism

When one calls a native routine using one of the `.C()`, `.Call()` or `.Fortran()` interface functions, one supplies the name of the native routine to invoke, the arguments to be passed to that routine and a non-obligatory `PACKAGE` argument identifying the DLL in which to search for the routine. The standard mechanism uses the operating system facilities to look in the DLL corresponding to the `PACKAGE` argument (or through all DLLs if the caller did not specify a value for the `PACKAGE` argument.) This lookup means that we can ask for *any* available routine in the library, whether it was intended to be called by the S programmer or internally by other routines in the DLL. Also, we know nothing about that routine: the number or type of arguments it expects, what it returns.

It is common to mistakenly invoke a routine designed for use with `.Call()`, but using the `.C()` func-

[1]The `.Call()` function allows one to pass regular S objects directly between S and C.

tion. On some machines, this this can crash R and one can lose the data in the R session. For example, on Solaris this will usually cause a crash but not on Linux. Or is it on Solaris only if one uses `gcc`? or Sun's own compilers? That's really the point: we are depending on highly system-specific features that are not entirely reproducible and can be very, very frustrating to diagnose. Ideally, we want S to help out and tell us we are calling native routines with the wrong function, signal that we have the wrong number of arguments, and perhaps even convert those arguments to the appropriate types.

Registering routines

Well, there is a better approach which allows S to do exactly these things. The idea is to have the DLL explicitly tell S which routines are available to S, and for which interface mechanisms (`.C()`, `.Call()`, ...). R stores the information about these routines and consults it when the user calls a native routine. When does the DLL get to tell R about the routines? When we load the DLL, R calls the R-specific initialization routine in that DLL (named `R_init_`*dllname*`()`), if it exists. This routine can register the routines as well as performing any other initialization it wants.

An example will hopefully make things clear. We will create a shared library named 'myRoutines.so'[2]. This provides two routines (*fooC()* and *barC()*) to be called via the `.C()` function and one (*myCall()*) to be accessed via `.Call()`. We'll ignore the code in the routines here since our purpose is only to illustrate how to register the routines.

```
static void fooC(void)
{ ... }

static void barC(double *x, Rint *len)
{ ... }

static SEXP myCall(SEXP obj)
{ return(obj); }
```

Now that we have defined these routines, we can add the code to register them (see figure 2). We create two arrays, one for each of the `.C()` and `.Call()` routines. The types of the arrays are *R_CMethodDef* and *R_CallMethodDef*, respectively. Each routine to be registered has an entry in the appropriate array. These entries (currently) have the same form for each type of routine and have 3 required elements:

S name The name by which S users refer to the routine. This does not have to be the same as the name of the C routine.

C routine This is the address of the routine, given simply by its name in the code. It should be cast to type *DL_FUNC*.

argument count The number of arguments the routine expects. This is used by R to check that the number of arguments passed in the call matches what is expected. In some circumstances one needs to avoid this check. Specifying a value of `-1` in this field allows this.

The last entry in each top-level array must be `NULL`. R uses this to count the number of routines being registered.

For our example, these arrays are defined in figure 1. The code includes the file 'R_ext/Rdynload.h' so as to get the definitions of the array types. Then we list the two entries for the `.C()` routines and the single entry in the *R_CallMethodDef* array.

```
#include <R_ext/Rdynload.h>

static const
R_CMethodDef cMethods[] = {
   {"foo",  (DL_FUNC) &fooC, 0},
   {"barC", (DL_FUNC) &barC, 2},
   NULL
};

static const
R_CallMethodDef callMethods[] = {
   {"myCall", (DL_FUNC) &myCall, 1},
   NULL
};
```

Figure 1: Defining the registration information

The very final step is to define the initialization routine that is called when the DLL is loaded by R. Since the DLL is called 'myRoutines.so', the name of the initialization routine is *R_init_myRoutines()*. When the DLL is loaded, R calls this with a single argument (`info`) which is used to store information about the DLL being loaded. So we define the routine as follows:

```
void R_init_myRoutines(DllInfo *info)
{
 /* Register the .C and .Call routines.
    No .Fortran() or .External() routines,
    so pass those arrays as NULL.
  */
 R_registerRoutines(info,
                    cMethods, callMethods,
                    NULL, NULL);
}
```

Figure 2: Registering the `.C()` and `.Call()` routines

From this point on, the library developer can proceed in the usual manner, and does not need to do

[2]The extension is platform-specific, and will '.dll' on Windows.

anything else for the registration mechanism. She compiles the library using the usual command and loads it using `dyn.load()` or `library.dynam()`. In my example, I have a single file named 'myRoutines.c' and, in Unix, create the DLL with the command

```
R CMD SHLIB myRoutines.c
```

The internal R code will determine whether the registration mechanism is being used and take the appropriate action.

Now we can test our example and see what the registration mechanism gives us. First, we start R and load the DLL. Then we call the routine *foo()*. Next, we intentionally call this with errors and see how R catches these.

```
> dyn.load("myRoutines.so")
> .C("foo")
In fooC
list()
> .C("foo", 1)
Error: Incorrect number of arguments (1),
   expecting 0 for foo
> .Call("foo")  # Should be .C("foo")
Error in .Call("foo") :
   .Call function name not in load table
```

Next, we move to the `.Call()` routine *myCall()*.

```
> .Call("myCall") # no argument
Error: Incorrect number of arguments (0),
   expecting 1 for myCall
> .Call("myCall", 1)
In myCall
[1] 1
> .C("myCall", 1) # Should be .Call("myCall")
Error in .C("myCall", 1) :
    C/Fortran function name not in load table
```

The very observant reader may have noticed that the three routines have been declared to be **static**. Ordinarily this would mean that they are not visible to R. Since we explicitly register the routines with R by their addresses (and not during compilation), this works as intended. The routines are only accessed directly from within the file. And now we have reduced the potential for conflicts between symbols in different libraries and of finding the wrong symbol.

Our example dealt with routines to be called via the `.C()` and `.Call()` functions. FORTRAN routines and those called via the `.External()` function are handled in exactly the same way, defining arrays for those routines. In our example, we specified `NULL` for the 3rd and 4th arguments in the call to *R_registerRoutines()* to indicate that we had no routines in either of these categories.

Rarely are libraries completely cast in stone. We occasionally add routines and want to be able to call them from R. To do this, one should register them and this merely involves adding them to the appropriate array which is passed in the *R_registerRoutines()* call. When one is developing the library, it can be inconvenient to have to remember to register routines each time we add them. Instead, it would be useful to be able to use the registration mechanism *and*, if the routine was not found there, to default to the dynamic lookup mechanism. This is easy to do from within the initialization routine for the DLL. In that routine, add the call

```
  R_usedDynamicSymbols(info, TRUE);
```

where `info` is the `DllInfo` object passed as argument to the initialization routine.

One can find additional examples of how to use the registration mechanism in the packages shipped with R itself (**ctest**, **mva**, ...). Also more technical overview of the mechanism with some annotated examples and more motivation is available at http://developer.r-project.org/DynamicCSymbols.pdf.

Extended applications

The motivation for developing the registration mechanism was to avoid the problems discussed at the beginning of this article. However, now that we have this mechanism in place, it turns out that we can make more use of it.

We have seen how we can ensure that routines are called via the correct interface. In other words, we check that `.C()` routines are not called via `.Call()`, and similarly for the other interfaces. Verifying the number of arguments is convenient, especially when the author of the DLL is actively developing the code and changing the number of arguments.

We can take this one step further by specifying the types of the expected arguments in `.C()` and `.Fortran()` routines.[3] For instance, in our example, we could give the types of the two parameters of *barC()*. We haven't yet finalized the details of this interface and so it is not part of R quite yet. However, it might look something like the following:

```
static const R_CMethodDef cMethods[] = {
 {"foo",  (DL_FUNC) &fooC, 0},
 {"barC", (DL_FUNC) &barC, 2,
               { REALSXP, INTSXP } },
 NULL
};
```

When the internal mechanism associated with the `.C()` function handles a call to *barC()* it can then check that the S objects passed in the `.C()` call correspond to these types. R can raise an error if it discovers an argument of the wrong type, or alternatively can convert it to the type the routine is expecting. This is a powerful facility that not only reduces errors, but also proves to be very useful for handling large, external datasets. Indeed, R 1.3.0 has

[3] This isn't as useful for `.Call()` and `.External()` since these take S objects which all have the same type.

a feature that allows users to specify conversion routines for certain types of objects that are handled via the .Call() (see http://cm.bell-labs.com/stat/duncan/SCConverters).

A potentially important use of the registration mechanism relates to security, and specifically prohibiting some users calling certain native routines that have access to sensitive data. We have been developing packages that embed R within spreadsheets such as Gnumeric and Excel; Web browsers such as Netscape; relational databases such as Postgres; and so on. One benefit of this approach is that one can run R code that is dynamically downloaded from the Web. However, as we all know, this is a common way to download viruses and generally make ones machine vulnerable. Using the registration mechanism, developers can mark their routines as being vulnerable and to be used only in "secure" sessions. What this means exactly remains to be defined!

Building the table automatically

This registration mechanism offers all the advantages that we have mentioned above. However, it requires a little more work by the developer. Since the original lookup mechanism still works, many developers may not take the time to create the arrays of routine definitions and register them. It would be convenient to be able to generate the registration code easily and without a lot of manual effort by the developer.

The **Slcc** (http://www.omegahat.org/Slcc/) package from the Omegahat project provides a general mechanism for processing C source code and returning information about the data structures, variables and routines it contains. This information is given as S objects and can be used to generate C code. The package provides a function to read both the S and C code of a library and generate the C code to register (only) the routines that are referenced in the S code.

The **Slcc** package is in the early stages of development. It runs on Linux, but there are some minor installation details to be worked out for other platforms.

Summary

The new registration mechanism is being used in the R packages within the core R distribution itself and seems to be working well. We hope some of the benefits are obvious. We expect that others will appear over time when we no longer have to deal with subtle differences in the behavior of various operating systems and how to handle dynamically loaded code. The only extra work that developers have to do is to explicitly create the table of routines that are to be registered with R. The availability of the **Slcc** package will hopefully help to automate the creation of the registration code and make it a trivial step. We are very interested in peoples' opinions and suggestions.

Duncan Temple Lang
Bell Labs, Murray Hill, New Jersey, U.S.A
duncan@research.bell-labs.com

Support Vector Machines

The Interface to libsvm in package e1071

by David Meyer

"Hype or Hallelujah?" is the provocative title used by Bennett & Campbell (2000) in an overview of Support Vector Machines (SVM). SVMs are currently a hot topic in the machine learning community, creating a similar enthusiasm at the moment as Artificial Neural Networks used to do before. Far from being a panacea, SVMs yet represent a powerful technique for general (nonlinear) classification, regression and outlier detection with an intuitive model representation.

Package **e1071** offers an interface to the award-winning[1] C++ SVM implementation by Chih-Chung Chang and Chih-Jen Lin, libsvm (current version: 2.31), featuring:

- *C*- and ν-classification
- one-class-classification (novelty detection)
- ϵ- and ν-regression

and includes:

- linear, polynomial, radial basis function, and sigmoidal kernels
- formula interface
- *k*-fold cross validation

For further implementation details on libsvm, see Chang & Lin (2001).

[1] The library won the IJCNN 2001 Challenge by solving two of three problems: the Generalization Ability Challenge (GAC) and the Text Decoding Challenge (TDC). For more information, see: http://www.csie.ntu.edu.tw/~cjlin/papers/ijcnn.ps.gz.

Basic concept

SVMs were developed by Cortes & Vapnik (1995) for binary classification. Their approach may be roughly sketched as follows:

Class separation: basically, we are looking for the optimal separating hyperplane between the two classes by maximizing the *margin* between the classes' closest points (see Figure 1)—the points lying on the boundaries are called *support vectors*, and the middle of the margin is our optimal separating hyperplane;

Overlapping classes: data points on the "wrong" side of the discriminant margin are weighted down to reduce their influence ("*soft margin*");

Nonlinearity: when we cannot find a *linear* separator, data points are projected into an (usually) higher-dimensional space where the data points effectively become linearly separable (this projection is realised via *kernel techniques*);

Problem solution: the whole task can be formulated as a quadratic optimization problem which can be solved by known techniques.

A program able to perform all these tasks is called a *Support Vector Machine*.

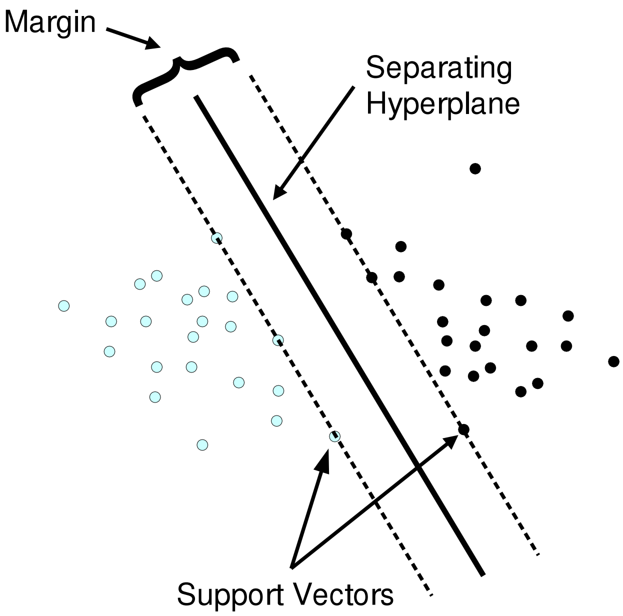

Figure 1: Classification (linear separable case)

Several extensions have been developed; the ones currently included in `libsvm` are:

ν-classification: this model allows for more control over the number of support vectors (see Schölkopf et al., 2000) by specifying an additional parameter ν which approximates the fraction of support vectors;

One-class-classification: this model tries to find the support of a distribution and thus allows for outlier/novelty detection;

Multi-class classification: basically, SVMs can only solve binary classification problems. To allow for multi-class classification, `libsvm` uses the *one-against-one* technique by fitting all binary subclassifiers and finding the correct class by a voting mechanism;

ϵ-regression: here, the data points lie *in between* the two borders of the margin which is maximized under suitable conditions to avoid outlier inclusion;

ν-regression: with analogous modifications of the regression model as in the classification case.

Usage in R

The R interface to `libsvm` in package **e1071**, `svm()`, was designed to be as intuitive as possible. Models are fitted and new data are predicted as usual, and both the vector/matrix and the formula interface are implemented. As expected for R's statistical functions, the engine tries to be smart about the mode to be chosen, using the dependent variable's type (y): if y is a factor, the engine switches to classification mode, otherwise, it behaves as a regression machine; if y is omitted, the engine assumes a novelty detection task.

Examples

In the following two examples, we demonstrate the practical use of `svm()` along with a comparison to classification and regression trees as implemented in `rpart()`.

Classification

In this example, we use the glass data from the UCI Repository of Machine Learning Databases (available in package **mlbench**) for classification. The task is to predict the type of a glass on basis of its chemical analysis. We start by splitting the data into a train and test set:

```
library(e1071)
library(rpart)
library(mlbench)
data(Glass)

## split data into a training and test set
index     <- 1:nrow(x)
testindex <- sample(index,
                    trunc(length(index)/3))
testset   <- x[testindex,]
trainset  <- x[-testindex,]
```

Both for the SVM and the partitioning tree (via `rpart()`), we fit the model and try to predict the test set values:

```
## svm
svm.model <- svm(Type ~ ., data = trainset,
                 cost = 100, gamma = 1)
svm.pred  <- predict(svm.model, testset[,-10])
```

(The dependent variable, `Type`, has column number 10. `cost` is a general parameter for C-classification and `gamma` is the radial basis function-specific kernel parameter.)

```
## rpart
rpart.model <- rpart(Type ~ ., data = trainset)
rpart.pred  <- predict(rpart.model,
                  testset[,-10], type = "class")
```

A cross-tabulation of the true versus the predicted values yields:

```
## compute svm confusion matrix
table(pred = svm.pred, true = testset[,10])

    true
pred 1  2 3 5 6 7
   1 8  7 2 0 0 0
   2 5 19 0 0 1 0
   3 3  3 2 0 0 0
   5 0  4 0 2 2 0
   6 0  0 0 0 3 0
   7 2  0 0 0 0 8

## compute rpart confusion matrix
table(pred = rpart.pred, true = testset[,10])

    true
pred 1  2 3 5 6 7
   1 8 10 2 2 2 0
   2 9 17 1 0 2 0
   3 0  4 1 0 0 0
   5 0  1 0 0 2 0
   6 0  0 0 0 0 0
   7 1  1 0 0 0 8
```

Finally, we compare the performance of the two methods by computing the respective accuracy rates and the kappa indices (as computed by `classAgreement()` also contained in package **e1071**). In Table 1, we summarize the results of 100 replications: `svm()` seems to perform slightly better than `rpart()`.

Non-linear ϵ-regression

The regression capabilities of SVMs are demonstrated on the ozone data, also contained in **mlbench**. Again, we split the data into a train and test set.

```
library(e1071)
library(rpart)
library(mlbench)
data(Ozone)

## split data into a training and test set
index     <- 1:nrow(x)
testindex <- sample(index,
                    trunc(length(index)/3))
testset   <- x[testindex,]
trainset  <- x[-testindex,]

## svm
svm.model <- svm(V4 ~ ., data = trainset,
                 cost = 1000, gamma = 0.0001)
svm.pred  <- predict(svm.model, testset[,-4])

## rpart
rpart.model <- rpart(V4 ~ ., data = trainset)
rpart.pred  <- predict(rpart.model, testset[,-4])
```

We compare the two methods by the mean squared error (MSE)—see Table 2. Here, in contrast to classification, `rpart()` does a better job than `svm()`.

Elements of the `svm` object

The function `svm()` returns an object of class "`svm`", which partly includes the following components:

`SV`: matrix of support vectors found;

`labels`: their labels in classification mode;

`index`: index of the support vectors in the input data (could be used e.g., for their visualization as part of the data set).

If the cross-classification feature is enabled, the `svm` object will contain some additional information described below.

Other main features

Class Weighting: if one wishes to weight the classes differently (e.g., in case of asymmetric class sizes to avoid possibly overproportional influence of bigger classes on the margin), weights may be specified in a vector with named components. In case of two classes A and B, we could use something like: `m <- svm(x, y, class.weights = c(A = 0.3, B = 0.7))`

Cross-classification: to assess the quality of the training result, we can perform a k-fold cross-classification on the training data by setting the parameter `cross` to k (default: 0). The `svm` object will then contain some additional values, depending on whether classification or regression is performed. Values for classification:

`accuracies`: vector of accuracy values for each of the k predictions

Index	Method	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
accuracy rate	svm	0.55	0.63	0.68	0.68	0.70	0.79
	rpart	0.49	0.63	0.65	0.66	0.70	0.79
kappa	svm	0.40	0.51	0.56	0.56	0.61	0.72
	rpart	0.33	0.49	0.52	0.53	0.59	0.70

Table 1: Performance of `svm()` and `rpart()` for classification (100 replications)

Method	Min.	1st Qu.	Median	Mean	3rd Qu.	Max.
svm	7.8	10.4	11.6	11.9	13.1	17.0
rpart	4.8	7.7	8.8	9.0	10.3	14.2

Table 2: Performance of `svm()` and `rpart()` for regression (mean squared error, 100 replications)

`tot.accuracy`: total accuracy

Values for regression:

`MSE`: vector of mean squared errors for each of the k predictions

`tot.MSE`: total mean squared error

`scorrcoef`: Squared correlation coefficient (of the predicted and the true values of the dependent variable)

Tips on practical use

- Note that SVMs may be very sensible to the proper choice of parameters, so always check a range of parameter combinations, at least on a reasonable subset of your data.

- For classification tasks, you will most likely use C-classification with the RBF kernel (default), because of its good general performance and the few number of parameters (only two: C and γ). The authors of `libsvm` suggest to try small and large values for C—like 1 to 1000—first, then to decide which are better for the data by cross validation, and finally to try several γ's for the better C's.

- Be careful with large datasets as training times may increase rather fast.

Conclusion

We hope that `svm` provides an easy-to-use interface to the world of SVMs, which nowadays have become a popular technique in flexible modelling. There are some drawbacks, though: SVMs scale rather badly with the data size due to the quadratic optimization algorithm and the kernel transformation. Furthermore, the correct choice of kernel parameters is crucial for obtaining good results, which practically means that an extensive search must be conducted on the parameter space before results can be trusted, and this often complicates the task (the authors of `libsvm` currently conduct some work on methods of efficient automatic parameter selection). Finally, the current implementation is optimized for the radial basis function kernel only, which clearly might be suboptimal for your data.

Bibliography

Bennett, K. P. & Campbell, C. (2000). Support vector machines: Hype or hallelujah? *SIGKDD Explorations*, **2**(2). `http://www.acm.org/sigs/sigkdd/explorations/issue2-2/bennett.pdf`. 23

Chang, C.-C. & Lin, C.-J. (2001). Libsvm: a library for support vector machines (version 2.31). `http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf`. 23

Cortes, C. & Vapnik, V. (1995). Support-vector network. *Machine Learning*, **20**, 1–25. 24

Schölkopf, B., Smola, A., Williamson, R. C., & Bartlett, P. (2000). New support vector algorithms. *Neural Computation*, **12**, 1207–1245. 24

Vapnik, V. (1998). *Statistical learning theory*. New York: Wiley.

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

A Primer on the R-Tcl/Tk Package

by Peter Dalgaard

Introduction

Tcl/Tk is a combination of a scripting language and a toolkit for graphical user interfaces. Since version 1.1.0, R has had a **tcltk** package to access the Tk toolkit, replacing Tcl code with R function calls (Dalgaard, 2001). There are still some design problems in it, but it is quite useful already in its current state.

This paper intends to get you started with the R-Tcl/Tk interface. Tcl/Tk is a large package, so it is only possible to explain the basic concepts here.

The presentation here is based on the X11/Unix version of R. The **tcltk** package also works on Windows. It (currently) does not work with Mac OS Classic but it does work with OS X. It is only the Linux variants that come with Tcl/Tk; on other systems some footwork will be necessary to get it installed.

Widgets

A *widget* is a GUI element. Tk comes with a selection of basic widgets: text editing windows, sliders, text entry fields, buttons, labels, menus, listboxes, and a canvas for drawing graphics. These can be combined to form more complex GUI applications.

Let us look at a trivial example:

```
library(tcltk)
tt <- tktoplevel()
lbl <- tklabel(tt, text="Hello, World!")
tkpack(lbl)
```

This will cause a window to be displayed containing the "Hello, World!" message. Notice the overall structure of creating a container widget and a child widget which is positioned in the container using a *geometry manager* (`tkpack`). Several widgets can be packed into the same container, which is the key to constructing complex applications out of elementary building blocks. For instance, we can add an "OK" button with

```
but <- tkbutton(tt, text="OK")
tkpack(but)
```

The window now looks as in Figure 1. You can press the button, but no action has been specified for it.

Figure 1: Window with label widget and button widget.

The title of the window is "1" by default. To set a different title, use

```
tktitle(tt) <- "My window"
```

Geometry managers

A geometry manager controls the placement of *slave* widgets within a *master* widget. Three different geometry managers are available in Tcl/Tk. The simplest one is called the *placer* and is almost never used. The others are the *packer* and the *grid* manager.

The packer "packs widgets in order around edges of cavity". Notice that there is a *packing order* and a *packing direction*.

In the example, you saw that the window autosized to hold the button widget when it was added. If you enlarge the window manually, you will see that the slave widgets are placed centered against the top edge. If you shrink it, you will see that the last packed item (the button) will disappear first. (Manual resizing disables autosizing. You can reenable it with `tkwm.geometry(tt,"")`.)

Widgets can be packed against other sides as well. A widget along the top or bottom is allocated a *parcel* just high enough to contain the widget, but occupying as much of the width of the container as possible, whereas widgets along the sides get a parcel of maximal height, but just wide enough to contain it. The following code may be illustrative (Figure 2):

```
tkdestroy(tt) # get rid of old example
tt <- tktoplevel()
edge <- c("top","right","bottom","left")
buttons <- lapply(1:4,
  function(i) tkbutton(tt, text=edge[i]))
for ( i in 1:4 )
  tkpack(buttons[[i]], side=edge[i],
         fill="both")
```

Figure 2: Geometry management by the packer

The `fill` argument causes each button to occupy its entire parcel. Similarly `expand=TRUE` causes parcels to increase in width or height (depending on the packing direction) to take up remaining space in the container. This occurs *after* allotment of parcels; in the above example only "left" can expand.

If an object does not fill its parcel, it needs to be *anchored*. The `anchor` argument to `tkpack` can be set to compass-style values like `"n"` or `"sw"` for placement in the middle top, respectively bottom left. The default is `"center"`.

It is useful at this point to consider what the packer algorithm implies for some typical layouts:

Simple vertical or horizontal stacking is of course trivial, you just keep packing against the same side.

For a text widget with a scrollbar on the side, you want `fill="y"` for the scrollbar and `fill="both"` and `expand=TRUE` for the text area. The scrollbar should be packed before the text widget so that the latter shrinks first.

A text widget with scrollbar and a row of buttons beneath it? You cannot do that with the packer algorithm! This is where *frames* come in. These are containers for further widgets with separate geometry management. So you pack the buttons inside a frame, pack the frame against the bottom, then pack the scrollbar and text widget.

The combination of the packer and frames gives a lot of flexibility in creating GUI layouts. However, some things are tricky, notably lining widgets up both vertically and horizontally.

Suppose you want multiple lines, each containing an entry widget preceded by a label. With the packer there is no simple way to keep the beginning of the entry fields lined up.

Enter the *grid manager*. As the name suggests it lays out widgets in rows and columns. Using this manager the labeled-entry problem could be solved as follows (Figure 3

```
t2 <- tktoplevel()
heading <- tklabel(t2, text="Registration form")
l.name <- tklabel(t2, text="Name")
l.age  <- tklabel(t2, text="Age")
e.name <- tkentry(t2, width=30)
e.age  <- tkentry(t2, width=3)

tkgrid(heading, columnspan=2)
tkgrid(l.name, e.name)
tkgrid(l.age, e.age)
tkgrid.configure(e.name, e.age, sticky="w")
tkgrid.configure(l.name, l.age, sticky="e")
```

Figure 3: A registration form using the grid manager

With the grid manager it is most convenient to specify a full row at the time, although options let you do it otherwise. The `columnspan` argument joins grid cells horizontally. The `sticky` argument works somewhat like anchoring in the packer. The value can be any subset of n, s, e, and w and specifies that the widget should stick to the specified sides of the cell. If it contains opposite sides, e.g. both n and s, the widget stretches to fill the space.

You can mix the geometry managers, although not in the same master frame. Some parts of an application may be best handled by the packer and others by the grid manager.

Communication with widgets

We need a way to get data from widgets to and from R, and a way to make things happen in response to widget events. There are two general patterns for this, namely *control variables* and *callbacks*.

Control variables associate the state of some aspect of a widget with a variable in Tcl. These Tcl variables can be accessed from R as (pseudo-)components of the `tclvar` object. So we could control the name entry field of the example above with

```
tkconfigure(e.name, textvariable="foo")
tclvar$foo <- "Hello, World"
```

and conversely any change to the content of the entry field is reflected in `tclvar$foo` *This mechanism is not optimal and will likely change in future versions of R!*

Control variables are also used by checkbuttons, radiobuttons, and scales. Radiobutton widgets allow a `value` argument so that the button lights up when the control variable has that value, and the variable is given that value when the radiobutton is clicked. A checkbutton will indicate whether its control variable is 0 (`FALSE`) or not.

Callbacks are functions that are linked to GUI events. Callbacks are often set up using arguments named `command`.

For a simple example consider

```
t3 <- tktoplevel()
b <- tkbutton(t3, text = "Don't press me!")
tkpack(b)
```

```
change.text <- function() {
  cat("OW!\n")
  tkconfigure(b, text = "Don't press me again!")
}
tkconfigure(b, command = change.text)
```

This callback function doesn't take any arguments, but others do. There are two ways to take account of this, depending on whether the callback is actively soliciting information or not. An example of the latter is the scrollbar protocol as exemplified below

```
t4 <- tktoplevel()
txt <- tktext(t4)
scr <- tkscrollbar(t4,
  command=function(...) tkyview(txt,...))
tkconfigure(txt,
  yscrollcommand=function(...) tkset(scr,...))
tkpack(scr, side="right", fill="y")
tkpack(txt, fill="both", expand=TRUE)
```

This sets up a bidirectional link: Manipulating the scrollbar changes the view of the text widget and vice versa. Some care is taken not to add a callback that refers to a widget before the widget exists.

We don't need to care what the arguments to the callbacks are, only to pass them through to `tkyview` and `tkset` respectively. In fact the arguments to `tkyview` will be different depending on which part of the scrollbar is engaged.

In Tcl, you can define a callback command as `myproc %x %y` and `myproc` will be invoked with the pointer coordinates as arguments. There are several other "percent codes". The parallel effect is obtained in R by defining the callback with specific formal arguments. From the `tkcanvas` demo:

```
plotMove <- function(x, y) {
  x <- as.numeric(x)
  y <- as.numeric(y)
  tkmove(canvas, "selected",
         x - lastX, y - lastY)
  lastX <<- x
  lastY <<- y
}
tkbind(canvas, "<B1-Motion>", plotMove)
```

The coordinates are passed as text strings, requiring the use of `as.numeric`.

Events and bindings

The previous example showed a binding of a callback to a windows event, containing the *event pattern* `<B1-Motion>` — mouse movement with Button 1 pressed.

An event pattern is a sequence of fields separated by hyphens and enclosed in `<>`. There are three kinds of fields, *modifier*, *type*, and *detail*, in that order. There can be several modifier fields. A generic example is `<Control-Alt-Key-c>`, where `Control` and `Alt` are modifiers, `Key` is the event type, and `c` is the detail. If `c` is left out any key matches. The `Key` part can be omitted when there's a character detail field. Similarly, a numeric detail field is assumed to refer to a button press event (notice that `<Key-1>` is different from `<1>`).

Callbacks are associated with events using `tkbind`, or sometimes `tktag.bind` or `tkitembind`.

Text widgets

The text widget in Tk embodies the functionality of a basic text editor, allowing you to enter and edit text, move around in the text with cursor control keys, and mark out sections of text for cut-and-paste operations. Here, we shall see how to add or delete text and how to extract the text of pieces thereof. These methods revolve around *indices*, *tags*, and *marks*.

A simple index is of the form `line.char` where `line` is the line number and `char` is the character position within the line. In addition there are special indices like `end` for the end of the text.

Tags provide a way of referring to parts of the text. The part of the text that has been marked as selected is tagged `sel`. Any tag can be used for indexing using the notation `tag.first` and `tag.last`.

Marks are somewhat like tags, but provide names for locations in the text rather than specific characters. The special mark `insert` controls and records the position of the insertion cursor.

To extract the entire content of a text widget, you say

```
X <- tkget(txt, "0.0", "end")
```

Notice that you have to give `0.0` as a character string, not as a number. Notice also that the result of `tkget` is a single long character string; you may wish to convert it to a vector of strings (one element per line) using `strsplit(X, "\n")`.

In a similar fashion, you can extract the selected part of the text with

```
X <- tkget(txt, "sel.first", "sel.last")
```

However, there is a pitfall: If there is no selection, it causes an error. You can safeguard against this by checking that

```
tktag.ranges(txt, "sel") != ""
```

Inserting text at (say) the end of a file is done with

```
tkinsert(txt, "end", string)
```

The string needs to be a single string just like the one obtained from `tkget`. If you want to insert an entire character array, you will need to do something along the lines of

```
tkinsert(txt, "end",
        paste(deparse(ls), collapse="\n"))
```

You can set the insertion cursor to the top of the text with

```
tkmark.set(txt, "insert", "0.0")
tksee(txt, "insert")
```

The `tksee` function ensures that a given index is visible.

An insertion leaves the insertion mark in place, but when it takes place exactly at the mark it is ambiguous whether to insert before or after the mark. This is controllable via *mark gravity*. The default is "right" (insert before mark) but it can be changed with

```
tkmark.gravity(txt, "insert", "left")
```

Creating menus

Tk menus are independent widgets. They can be used as popup menus, but more often they attach to the menu bar of a toplevel window, a menubutton, or a *cascade entry* in a higher-level menu.

Menus are created in several steps. First you setup the menu with `tkmenu`, then you add items with `tkadd`. There are so many possible options for a menu item that this is a more practicable approach.

Menu items come in various flavours. A *command entry* is like a button widget and invokes a callback function. *Cascade entries* invoke secondary menus. *Checkbutton* and *radiobutton* entries act like the corresponding widgets and are used for optional selections and switches. Special entries include *separators* which are simply non-active dividing lines and *tear-offs* which are special entries that you can click to detach the menu from its parent. The latter are on by default but can be turned off by passing `tearoff=FALSE` to `tkmenu`.

Here is a simple example of a menubutton with a menu which contains three radiobutton entries:

```
tclvar$color<-"blue"
tt <- tktoplevel()
tkpack(mb <- tkmenubutton(tt, text="Color"))
m <- tkmenu(mb)
tkconfigure(mb,menu=m)
for ( i in c("red", "blue", "green"))
  tkadd(m, "radio", label=i, variable="color",
        value=i)
```

A simple application: Scripting widgets

The following code is a sketch of a scripting widget (Figure 4. The widget can be used to edit multiple lines of code and submit them for execution. It can load and save files using `tk_getOpenFile` and `tk_getSaveFile`. For simplicity, the code is executed with `parse` and `eval`.

Notice that `tkcmd` is used to call Tcl commands that have no direct R counterpart. Future versions of the **tcltk** package may define functions `tkclose`, etc.

Tcl has file functions that by and large do the same as R connections do although they tend to work a little better with other Tcl functions.

You may want to experiment with the code to add features. Consider e.g. adding an Exit menu item, or binding a pop-up menu to Button 3.

```
tkscript <- function() {
  wfile <- ""
  tt <- tktoplevel()
  txt <- tktext(tt, height=10)
  tkpack(txt)
  save <- function() {
    file <- tkcmd("tk_getSaveFile",
      initialfile=tkcmd("file", "tail", wfile),
      initialdir=tkcmd("file", "dirname", wfile))
    if (!length(file)) return()
    chn <- tkcmd("open", file, "w")
    tkcmd("puts", chn, tkget(txt,"0.0","end"))
    tkcmd("close", chn)
    wfile <<- file
  }
  load <- function() {
    file <- tkcmd("tk_getOpenFile")
    if (!length(file)) return()
    chn <- tkcmd("open", file, "r")
    tkinsert(txt, "0.0", tkcmd("read", chn))
    tkcmd("close", chn)
    wfile <<- file
  }
  run <- function() {
    code <- tkget(txt,"0.0","end")
    e <- try(parse(text=code))
    if (inherits(e, "try-error")) {
      tkcmd("tk_messageBox",
            message="Syntax error",
            icon="error")
      return()
    }
    cat("Executing from script window:",
        "-----", code, "result:",  sep="\n")
    print(eval(e))
  }
  topMenu <- tkmenu(tt)
  tkconfigure(tt, menu=topMenu)
  fileMenu <- tkmenu(topMenu, tearoff=FALSE)
  tkadd(fileMenu, "command", label="Load",
        command=load)
  tkadd(fileMenu, "command", label="Save",
        command=save)
  tkadd(topMenu, "cascade", label="File",
        menu=fileMenu)
  tkadd(topMenu, "command", label="Run",
        command=run)
}
```

Further information

Some further coding examples are available in the demos of the **tcltk** package.

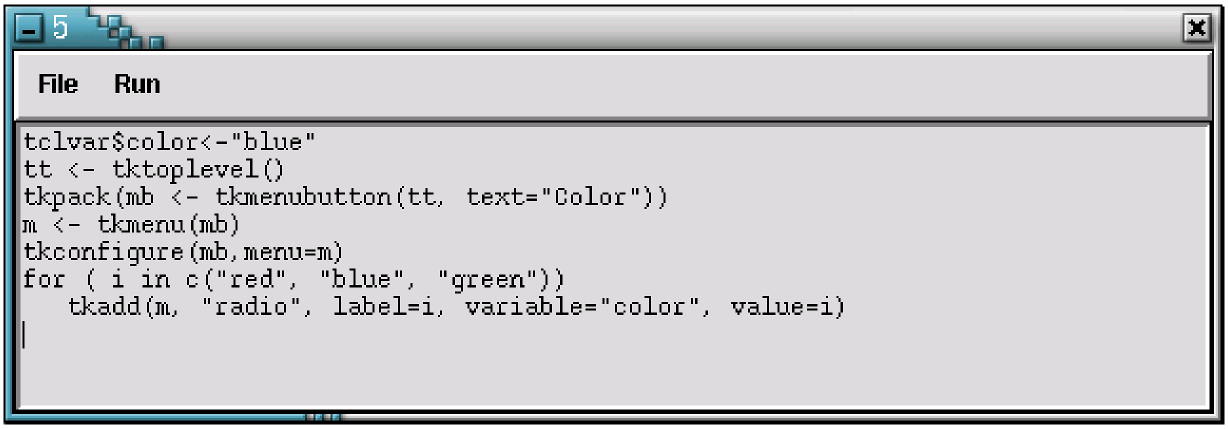

Figure 4: A simple scripting widget.

Most of the functions in the **tcltk** package are really just a thin layer covering an underlying Tcl command. Converting all the Tcl/Tk documentation for R is a daunting task, so you have to make do with the help for Tcl/Tk itself. This is fairly easy once you get the hang of some simple translation rules.

For the `tkbutton` function, you would look at the help for `button`. The R functions add a `tk` prefix to avoid name clashes. The `button` command in Tcl/Tk has a `-text` argument followed by the text string to display. Such options are replaced in the R counterpart by named arguments like `text="B1"`. The argument value is a string, but logical or numerical values, as well as (callback) functions are automatically converted.

When translating commands, there are a couple of special rules which are briefly outlined below.

One general difference is that Tcl encodes the widget hierarchy in the name of the widgets so that widget `.a` has subwidgets `.a.b` and `.a.c`, etc. This is impractical in R so instead of Tcl's

```
button .a.b -text foo
```

we specify the parent directly in the widget creation call

```
but <- tkbutton(parent, text="foo")
```

This pattern is used for all commands that create widgets. Another difference is that Tcl has *widget commands* like

```
.a.b configure -text fum
```

which in R is replaced by a command acting on a widget

```
tkconfigure(but, text="fum")
```

Some widget commands have subcommands as in

```
.a.b selection clear 0 end
```

which are turned into separate functions

```
tkselection.clear(lb, 0, "end")
```

In a few cases, the translation rules create ambiguities — for instance there is both a general `bind` command and a `bind` widget command for canvases. This has been resolved by making the widget commands `tkitembind`.

There is quite a large literature on Tcl and Tk. A well-reputed book is Welch (2000). A smaller reference item is Raines and Tranter (1999), although it is mostly a paper copy of online information. The main web site is at http://tcl.activestate.com/.

The useful `tclhelp` program comes with the TclX package. There is also a nice widget demo in the Tk distribution.

Bibliography

Peter Dalgaard. The R-Tcl/Tk interface. In Kurt Hornik and Fritz 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. 27

Paul Raines and Jeff Tranter. *Tcl/Tk in a Nutshell*. O'Reilly, 1999. 31

Brent B. Welch. *Practical Programming in Tcl and Tk*. Prentice-Hall PTR, New Jersey, 3rd edition, 2000. 31

Peter Dalgaard
University of Copenhagen, Denmark
P.Dalgaard@biostat.ku.dk

wle: A Package for Robust Statistics using Weighted Likelihood

by Claudio Agostinelli

The **wle** is a package for robust statistics using the weighted-likelihood estimating equations approach. This approach is different in many aspects from that presented in Huber (1981) and Hampel et al. (1986). It provides a general framework so that extensions are simpler than in the classical setting. The main feature is to provide first-order efficient (asymptotically) and robust estimators in the sense of breakdown. The current version (0.6-1) of the package implements most of the results presented in the literature.

In the next section we will introduce the weighted likelihood methodology and review the present literature. In the section *Package features* we will give some details about the current release and we will provide an example of some functions.

Weighted likelihood

The definition of Weighted Likelihood Estimating Equations (WLEE) was first proposed by Markatou et al. (1997) for discrete distributions, then in Markatou et al. (1998) the methods were extended to continuous models.

Let $x_1, x_2, \ldots, x_n$ be an i.i.d. sample from the random variable X with unknown density $f(\cdot)$ corresponding to the probability measure $F(\cdot)$. We will use the density $m(\cdot;\theta)$ corresponding to the probability measure $M(\cdot;\theta)$ and $\theta \in \Theta$ as a model for the random variable X. Note that in the maximum-likelihood context we assume $f(\cdot) \equiv m(\cdot;\theta_T)$ (almost surely) and $\theta_T \in \Theta$. Let $u(x;\theta) = \frac{\partial}{\partial\theta}\log m(x;\theta)$ be the score function. Under regularity conditions the maximum likelihood estimator of θ is a solution of the likelihood equation $\sum_{i=1}^n u(x_i;\theta) = 0$.

Given any point x in the sample space, Markatou et al. (1998) construct a weight function $w(x;\theta,\hat{F}_n)$ that depends on the chosen model distribution M and the empirical cumulative distribution $\hat{F}_n(t) = \sum_{i=1}^n \mathbf{1}_{x_i<t}/n$. Estimators for the parameter vector θ are obtained as solutions to the set of estimating equations:

$$\sum_{i=1}^n w(x_i;\theta,\hat{F}_n)u(x_i;\theta) = 0 \qquad (1)$$

The weight function

$$w(x;\theta,\hat{F}_n) = \min\left\{1, \frac{[A(\delta(x;\theta,\hat{F}_n))+1]^+}{\delta(x;\theta,\hat{F}_n)+1}\right\}$$

(where $[\cdot]^+$ indicates the positive part) takes values in the interval $[0,1]$ by construction.

The quantity $\delta(x;\theta,\hat{F}_n)$ is called the *Pearson residual*, defined as $\delta(x;\theta,\hat{F}_n) = f^*(x)/m^*(x;\theta) - 1$, where $f^*(x) = \int k(x;t,h)\, d\hat{F}_n(t)$ is a kernel density estimator and $m^*(x;\theta) = \int k(x;t,h)\, dM(t;\theta)$ is the smoothed model density. Note that sometimes $f^*(x)$ is a function of θ as in the regression case. The Pearson residual expresses the agreement between the data and the assumed probability model. The function $A(\cdot)$ is a *residual adjustment function*, RAF, (Lindsay, 1994) and it operates on Pearson residuals in the same way as the Huber ψ-function operates on the structural residuals. When $A(\delta) = \delta$ we have $w(x;\theta,\hat{F}_n) \equiv 1$, and this corresponds to maximum likelihood. Generally, the weights w use functions $A(\cdot)$ that correspond to a minimum disparity problem. For example, the function $A(\delta) = 2\{(\delta+1)^{1/2} - 1\}$ corresponds to Hellinger distance. For an extensive discussion of the concept of RAF see Lindsay (1994).

This weighting scheme provides first-order efficient (asymptotically) and robust estimators in the sense of breakdown, provided that one selects a root by using the parallel disparity measure (Markatou et al., 1998). However, the inspection of all roots is useful for diagnostics and data analysis.

The estimating equations (1) are solved using a re-weighting scheme. An algorithm based on resampling techniques is used to identify the roots and to provide starting values. Sub-samples of fixed dimension and without replication are sampled from the dataset. From each of these sub-samples a maximum likelihood estimator is evaluated and used to start the re-weighted algorithm.

To calculate the Pearson residuals we need to select the smoothing parameter h. Markatou et al. (1998) select $h^2 = g\sigma^2$, where g is a constant independent of the scale of the model which is selected in a way such that it assigns a very small weight to an outlying observation (Agostinelli and Markatou, 2001).

To illustrate the behaviour of the weight function, let us consider its asymptotic value when the data come from a mixture of two normal distributions, $f(x) = 0.9\mathrm{N}(0,1) + 0.1\mathrm{N}(4,1)$ (Figure 1). We set $g = 0.003$ with a normal kernel and we use a location normal family ($\mathrm{M} = \{\mathrm{N}(\theta,1),\ \theta \in \mathcal{R}\}$) as a model for these data. In Figure 2 we report the Pearson residuals evaluated in the distribution of the majority of the data, that is in $\theta = 0$, while in Figure 3 we report the corresponding weight function based on the Hellinger Residual Adjustment Function.

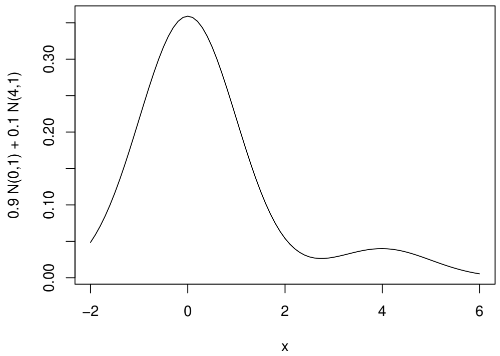

Figure 1: The contaminated normal density distribution

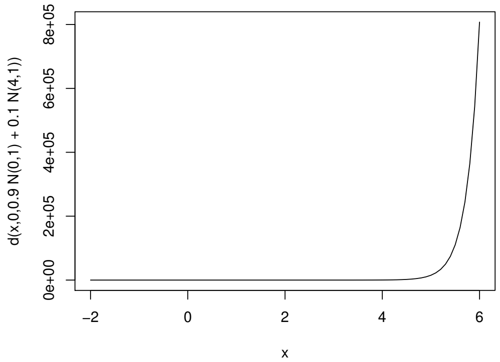

Figure 2: The (asymptotic) Pearson residuals.

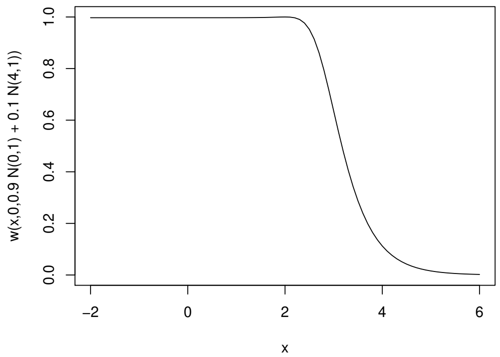

Figure 3: The (asymptotic) weights based on Hellinger Residual Adjustment Function.

Agostinelli (1998a,b) extended the methodology to the regression model while Agostinelli and Markatou (1998) studied the one-step estimator based on high breakdown initial estimator. Agostinelli (2000, 2001a,b) developed robust model selection procedures based on weighted versions of Akaike Information Criterion, Mallows C_p, Cross-Validation and Stepwise. Agostinelli (1998a, 2001d) and Agostinelli and Markatou (2001) defined weighted versions of the classical likelihood test functions: likelihood ratio, Wald and Rao (or score) tests. Markatou (2000, 2001) studied the estimation problem in a mixture model. Agostinelli (2001c) proposed estimation procedures for ARMA models.

Package features

Version 0.6-1 of package **wle** implements almost all the methods and procedures presented above. In particular, there are functions providing estimates the parameters for the *binomial* and *Poisson* models, and the univariate and multivariate *normal* and *gamma* models. A function is devoted to the *regression* model with normal errors and model-selection procedures are available for this case. The *weighted t-test* is available for one and two samples (paired and unpaired), with a function that works very similarly to the `t.test` function. The *weighted F-test* (Agostinelli, 2001b) may be used for comparison of two variances for objects generated by `wle.normal` and `wle.lm` in the same way as the `var.test` function. Finally, a preliminary version for estimating the parameters of a *univariate normal mixture* models is available.

In the following example we illustrate the functions `wle.lm` and `wle.cv` together with the related methods. We generated a dataset of 70 observations. The first 60 observations follow the $Y = 8\ \log(X+1) + \varepsilon$ regression model with $\varepsilon \sim \mathrm{N}(0, 0.6)$ while the last 10 observations are a cluster of observations from the same model but with residuals from $\varepsilon \sim \mathrm{N}(-4, 0.6)$. The contamination level is about 14%.

```
> library(wle)
> set.seed(1234)

> x.data <- c(runif(60,20,80), runif(10,73,78))
> e.data <- rnorm(70,0,0.6)
> y.data <- 8*log(x.data+1)+e.data
> y.data[61:70] <- y.data[61:70] - 4
>
> x.model <- function(x) 8*log(x+1)
> x.log.data <- x.model(x.data)/8
```

First, we show how the `wle.lm` works: see Figure 4 on page 35. The function uses `formula` for describing the regression model structure as in `lm`. The most important parameters are `boot`, `group` and `num.sol`. The first parameter is used to control the number of bootstrap sub-samples, i.e., the number of starting values the function has to use in order to look for different roots of the estimating equation.

The second parameter is the size of the subsamples; it can not be less than the number of the unknown parameters since we have to obtain maximum likelihood estimates from those sub-samples. Markatou et al. (1998) empirically found that in most cases, it was sufficient to let `group` equal to the number of parameters to be estimated in order to produce reasonable estimates. On the other hand, in particular cases, this could raise some problems, for instance in the presence of highly correlated explanatory variables. For this reason the default value is the maximum of the number of parameters and one quarter of the sample size. In our example we set `group` equal the number of parameters.

The third parameter `num.sol` controls the maximum number of roots we expect to find. The algorithm is stopped when it has found `num.sol` roots, regardless of the number of bootstrap replications. Two roots are considered to be distinct if they have at least one component bigger than the `equal` parameter in absolute difference.

The `wle.lm` function has `summary` and `plot` methods. The summary is very similar to that generated for `lm`: one summary is reported for each root. `wle.lm` has found the "robust" root and the MLE-like root. The t-test is performed accordingly with the weighted Wald test for each root (Agostinelli, 2001d; Agostinelli and Markatou, 2001). Moreover, the weighted residuals are obtained as $weights * residuals$ without squaring the weights. In Figure 5 we present the dataset, the true model, the two models suggested by the weighted likelihood and the one found by maximum likelihood.

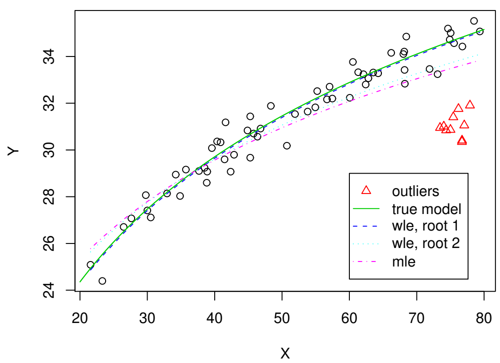

Figure 5: The dataset and the estimated model by `wle.lm` and `lm`.

The `plot` method helps to understand the difference between the roots. The first plot represents in the main diagonal the weights related to each root. The observations with weights close to one are shown in green, while the possible outliers with weights close to zero are displayed in red. The threshold is set by `level.weight` (with default value 0.5). In the lower triangle we compare the weight given by different roots to each observation while in the upper triangle the (unweighted) residuals are compared; the bisector is reported.

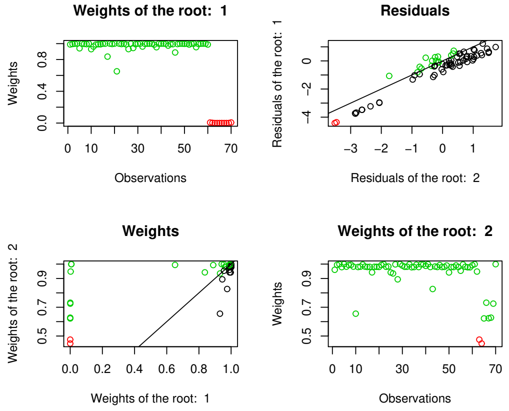

Figure 6: Plot from the `plot` method.

Then, for each root we present the qq-norm of the unweighted and weighted residuals and residuals vs fitted values plots. The observations with weights less than `level.weight` are reported in red.

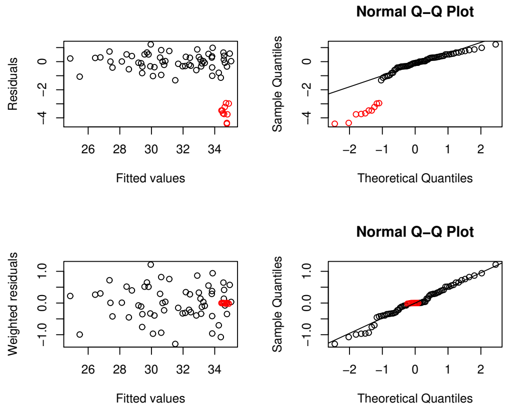

Figure 7: Plot from the `plot` method, root 1.

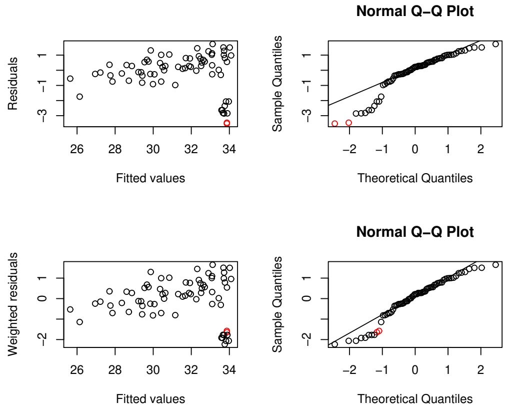

Figure 8: Plot from the `plot` method, root 2.

Now, we try to identify a good model for our dataset given a set of possible explanatory variables that include the "true" ones. Such variables are

```
> wle.lm.result <-
+ wle.lm(y.data~x.log.data, boot=50,
+        group=3, num.sol=3)
> summary(wle.lm.result)

Call:
wle.lm(formula = y.data ~ x.log.data, boot = 50,
       group = 3, num.sol = 3)

Root  1

Weighted Residuals:
     Min       1Q   Median       3Q      Max
-1.30752 -0.32307 -0.04171  0.32204  1.21939

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept) -0.09666    0.90864  -0.106    0.916
x.log.data   8.00914    0.23108  34.660   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01
                ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5705 on 56.80948 degrees of freedom
Multiple R-Squared: 0.9548,
Adjusted R-squared: 0.9452
F-statistic:  1201 on 1 and 56.80948
degrees of freedom, p-value: 0


Call:
wle.lm(formula = y.data ~ x.log.data, boot = 50,
       group = 3, num.sol = 3)

Root  2

Weighted Residuals:
    Min      1Q  Median      3Q     Max
-2.3863 -0.4943  0.2165  0.7373  1.6909

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)   4.9007     1.7282   2.836  0.00612 **
x.log.data    6.6546     0.4338  15.340  < 2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01
                ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 1.151 on 63.58554 degrees of freedom
Multiple R-Squared: 0.7873,
Adjusted R-squared: 0.7692
F-statistic: 235.3 on 1 and 63.58554
degrees of freedom, p-value: 0
```

Figure 4: Using `wle.lm`

highly correlated, and a model with just one of them could be a good model.

```
> xx <- cbind(x.data,x.data^2,x.data^3,
+       log(x.data+1))
> colnames(xx) <- c("X","X^2","X^3","log(X+1)")
> cor(xx)
                 X       X^2       X^3  log(X+1)
X        1.0000000 0.9913536 0.9718489 0.9879042
X^2      0.9913536 1.0000000 0.9942109 0.9595103
X^3      0.9718489 0.9942109 1.0000000 0.9253917
log(X+1) 0.9879042 0.9595103 0.9253917 1.0000000
```

We address the problem by using `wle.cv` which performs Weighted Cross-Validation. For comparison we use `mle.cv` which performs the classical Cross-Validation procedure (Shao, 1993). Since the procedure uses weights based on the full model, a crucial problem arises when multiple roots are present in this model. Currently the package chooses the root with the smallest scale parameter: this should work fine in most cases. In the next release of the package we will give the users the opportunity to choose the root by themselves. As seen in Figure 9 on page 37, while `wle.cv` suggests the "true" model, `mle.cv` chooses models with three explanatory variables.

Next, we estimate the suggested model; only one root is found (see Figure 10 on page 38). In Figure 11 we report the suggested models by the weighted likelihood and the classical procedure.

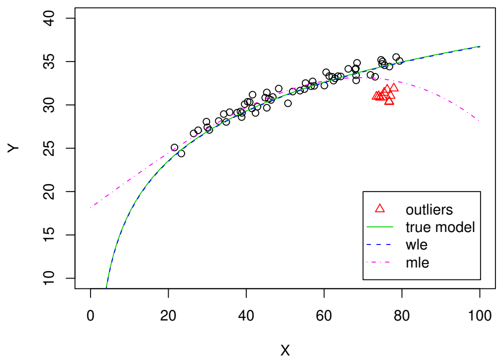

Figure 11: The model suggested by `wle.cv` and `mle.cv`.

Future developments

The next version of the package will probably include functions for autoregressive time-series. In particular, there will be functions for seasonal ARI models, with weighted autocorrelation functions, unit-root tests and perhaps a model selection procedure based on weighted Akaike Information Criterion for the order of the ARI model. Functions will be provided for the logistic regression model. Moreover, we will start to prepare a document to illustrate the use of the functions in applications.

Bibliography

C. Agostinelli. *Inferenza statistica robusta basata sulla funzione di verosimiglianza pesata: alcuni sviluppi*. PhD thesis, Dipartimento di Scienze Statistiche, Università di Padova, 1998a. 33

C. Agostinelli. Verosimiglianza pesata nel modello di regressione lineare. In *XXXIX Riunione scientifica della Società Italiana di Statistica*, Sorrento, 1998b. 33

C. Agostinelli. Robust model selection by cross-validation via weighted likelihood methodology. Submitted to *Australian and New Zealand Journal of Statisitcs*, 2000. 33

C. Agostinelli. Robust model selection in regression via weighted likelihood methodology. To appear in *Statistics & Probability Letters*, 2001a. 33

C. Agostinelli. Robust stepwise regression. To appear in *Journal of Applied Statistics*, 2001b. 33

C. Agostinelli. Robust time series estimation via weighted likelihood. Presented to the first International Conference on Robust Statistics, (poster session), Vorau, Austria, 2001c. 33

C. Agostinelli. Un approccio robusto alla verifica d'ipotesi basato sulla funzione di verosimiglianza pesata – robust testing hypotheses via weighted likelihood function. To appear in *Statistica*, 2001d. In Italian. 33, 34

C. Agostinelli and M. Markatou. A one–step robust estimator for regression based on the weighted likelihood reweighting scheme. *Statistics & Probability Letters*, 37(4):341–350, 1998. 33

C. Agostinelli and M. Markatou. Test of hypotheses based on the weighted likelihood methodology. *Statistica Sinica*, 11(2):499–514, 2001. 32, 33, 34

F. R. Hampel, E. M. Ronchetti, P. J. Rousseeuw, and W. A. Stahel. *Robust Statistics: The Approach based on Influence Functions*. John Wiley, New York, 1986. 32

P. J. Huber. *Robust Statistics*. John Wiley, New York, 1981. 32

B. G. Lindsay. Efficiency versus robustness: The case for minimum hellinger distance and related methods. *Annals of Statistics*, 22:1018–1114, 1994. 32

M. Markatou. Mixture models, robustness and the weighted likelihood methodology. *Biometrics*, 56: 483–486, 2000. 33

```
> wle.cv.result <- wle.cv(y.data~xx, boot=50,
+                         group=6, num.sol=3)
> summary(wle.cv.result, num.max=10)

Call:
wle.cv(formula = y.data ~ xx, boot = 50,
                  group=6, num.sol = 3)
     (Intercept) xxX xxX^2 xxX^3 xxlog(X+1)    wcv
 [1,]           0   0     0     0          1 0.3395
 [2,]           0   1     0     0          1 0.3631
 [3,]           0   0     1     0          1 0.3632
 [4,]           0   0     0     1          1 0.3635
 [5,]           1   0     0     0          1 0.3639
 [6,]           0   0     1     1          1 0.3868
 [7,]           0   1     0     1          1 0.3881
 [8,]           0   1     1     0          1 0.3896
 [9,]           1   0     0     1          1 0.3925
[10,]           1   0     1     0          1 0.3951

Printed the first  10  best models

> mle.cv.result <- mle.cv(y.data~xx)
> summary(mle.cv.result, num.max=10)

Call:
mle.cv(formula = y.data ~ xx)


Cross Validation selection criteria:
     (Intercept) xxX xxX^2 xxX^3 xxlog(X+1)    cv
 [1,]           1   1     0     1          0 1.557
 [2,]           1   1     1     0          0 1.560
 [3,]           0   0     1     1          1 1.579
 [4,]           0   1     0     1          1 1.581
 [5,]           0   0     0     1          1 1.584
 [6,]           0   1     1     0          1 1.589
 [7,]           1   0     1     1          0 1.593
 [8,]           1   0     0     1          1 1.594
 [9,]           1   0     1     0          1 1.617
[10,]           0   0     1     0          1 1.620

Printed the first  10  best models
```

Figure 9: Finding a good model using `wle.cv` and `mle.cv`

```
> wle.lm.result.cv <- wle.lm(y.data~x.log.data
+      -1, boot=50, group=3, num.sol=3)
> summary(wle.lm.result.cv)

Call:
wle.lm(formula = y.data ~ x.log.data - 1,
       boot = 50, group = 3, num.sol = 3)

Root  1

Weighted Residuals:
     Min       1Q   Median       3Q      Max
-1.30476 -0.32233 -0.03861  0.32276  1.21646

Coefficients:
           Estimate Std. Error t value Pr(>|t|)
x.log.data  7.98484    0.01874   426.1   <2e-16 ***
---
Signif. codes: 0 ‘***’ 0.001 ‘**’ 0.01
               ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.5649 on 57.76524 degrees of freedom
Multiple R-Squared: 0.9997,
Adjusted R-squared: 0.9996
F-statistic: 1.815e+05 on 1 and 57.76524
degrees of freedom, p-value: 0
```

Figure 10: Fitting the model suggested by weighted cross-validation.

M. Markatou. A closer look at the weighted likelihood in the context of mixtures. In C. A. Charalambides, M. V. Koutras, and N. Balakrishnan, editors, *Probability and Statistical Models with Applications*, pages 447–467. Chapman and Hall/CRC, 2001. 33

M. Markatou, A. Basu, and B. G. Lindsay. Weighted likelihood estimating equations: The discrete case with applications to logistic regression. *Journal of Statistical Planning and Inference*, 57:215–232, 1997. 32

M. Markatou, A. Basu, and B. G. Lindsay. Weighted likelihood estimating equations with a bootstrap root search. *Journal of the American Statistical Association*, 93:740–750, 1998. 32, 34

J. Shao. Linear model selection by cross-validation. *Journal of the American Statistical Association*, 88: 486–494, 1993. 36

Claudio Agostinelli
Dipartimento di Statistica
Università Cà Foscari di Venezia
30125, Venezia
claudio@unive.it

Changes on CRAN

by Kurt Hornik and Friedrich Leisch

CRAN packages

The following extension packages from ‘src/contrib’ were added since the last newsletter.

AnalyzeFMRI Functions for I/O, visualisation and analysis of functional Magnetic Resonance Imaging (fMRI) datasets stored in the ANALYZE format. By J L Marchini.

EMV Estimation of missing values in a matrix by a k-th nearest neighbors algorithm. By Raphael Gottardo.

Rwave Rwave is a collection of R functions which provide an environment for the Time-Frequency analysis of 1-D signals (and especially for the wavelet and Gabor transforms of noisy signals). It is based on the book: ‘Practical Time-Frequency Analysis: Gabor and Wavelet Transforms with an Implementation in S’, by Rene Carmona, Wen L. Hwang and

Bruno Torresani, Academic Press (1998). S original by Rene Carmona, R port by Brandon Whitcher.

car Contains mostly functions for applied regression, linear models, and generalized linear models, with an emphasis on regression diagnostics, particularly graphical diagnostic methods. By John Fox.

diamonds Functions for illustrating aperture-4 diamond partitions in the plane, or on the surface of an octahedron or icosahedron, for use as analysis or sampling grids. By Denis White.

fastICA Implementation of FastICA algorithm to perform Independent Component Analysis (ICA) and Projection Pursuit. By J L Marchini and C Heaton.

fields A collection of programs for curve and function fitting with an emphasis on spatial data. The major methods implemented include cubic and thin plate splines, universal Kriging and Kriging for large data sets. The main feature is that any covariance function implemented in R can be used for spatial prediction. By Doug Nychka.

pcurve Fits a principal curve to a numeric multivariate dataset in arbitrary dimensions. Produces diagnostic plots. Also calculates Bray-Curtis and other distance matrices and performs multi-dimensional scaling and principal component analyses. S original by Trevor Hastie, S+ library by Glenn De'ath, R port by Chris Walsh.

pixmap Functions for import, export, plotting and other manipulations of bitmapped images. By Friedrich Leisch and Roger Bivand.

rpvm Provides interface to PVM APIs, and examples and documentation for its use. By Na (Michael) Li and A. J. Rossini.

sem Contains functions for fitting general linear structural equation models (with observed and unobserved variables) by the method of maximum likelihood using the RAM approach, and for fitting structural equations in observed-variable models by two-stage least squares. By John Fox.

sptests A collection of tests for spatial autocorrelation, including global Moran's I and Geary's C. By Roger Bivand.

spweights 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. By Roger Bivand and Nicholas Lewin-Koh.

vegan Various help functions for community ecologists. By Jari Oksanen.

waveslim Basic wavelet routines for time series analysis, based on wavelet methodology developed in 'Wavelet Methods for Time Series Analysis', by D. B. Percival and A. T. Walden, Cambridge University Press (2000), along with 'An Introduction to Wavelets and Other Filtering Methods in Finance and Economics' by R. Gencay, F. Selcuk and B. Whitcher, Academic Press (2001). By Brandon Whitcher.

CRAN mirrors the R packages from the Omegahat project in directory '`src/contrib/Omegahat`'. The following are recent additions:

SASXML Example for reading XML files in SAS 8.2 manner. By Duncan Temple Lang.

Sxslt An extension module for libxslt, the XML-XSL document translator, that allows XSL functions to be implemented via R functions.

Checking packages

The current development version of R (the forthcoming 1.4.0) features a much more sophisticated test suite for checking packages with the `R CMD check` utility. Especially the checks for consistency between code and documentation are much better, and we have started to use these checks for all contributions to CRAN. Several contributors to CRAN already had the frustrating experience that their package passed `R CMD check` on their machine (running 1.3.1) without a warning, and we responded along the lines of *"thanks for your contribution to the R project, but perhaps you find some time to fix ..."*.

We want to keep the quality of R as high as possible, and with that we mean the whole community effort, not only the base system. R would not be what it is today without all those wonderful packages contributed to CRAN. As mentioned above, the new suite of checks will be released as part of R 1.4.0, in the meantime we would like to invite all package developers to download a CVS snapshot of the development version and run it from there.

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

Friedrich Leisch
Technische Universität Wien, Austria
`Friedrich.Leisch@ci.tuwien.ac.at`

Changes in R

by the R Core Team

New features in version 1.3.1

- `massage-examples` is now a Perl script and about 50x faster.
- On Unix(-alike) systems the default pager is now determined during configuration, and is 'less' if available, otherwise 'more' (and not 'more -s' as previously).
- configure now tests for strptime functions that fail on inputs before 1970 (found on Irix). It no longer checks for the SCSL and SGIMATH libraries on Irix.
- New formula interface to `cor.test()` in package **ctest**.
- `"NA"` is now a valid color name (as `NA` has been a valid integer color).
- `pairs()` function has a new 'gap' argument for adjusting the spacing between panels.
- `R CMD check` has a new test for unbalanced braces in Rd files.
- `readBin()` has a new argument 'signed' to simplify reading unsigned 8- and 16-bit integers.
- New `capabilities()` option `"cledit"`.
- Modified restore code to give clearer error messages in some cases.

New development model

Previously, there were two development versions of R: one for fixing bugs in the current release ('stable') and one for adding new features ('unstable'). This two-tier model, which has been successful for some open source projects, did not optimally meet the needs of the R Core development team. Hence, as of the release of R 1.3.1, there are now three development versions of R, working towards the next patch ('r-patched'), minor ('r-devel'), and major ('r-ng') releases of R, respectively. Version r-patched is for bug fixes mostly. New features are typically introduced in r-devel. Version r-ng will eventually become the next generation of R. The three versions correspond to the R 'major.minor.patchlevel' numbering scheme.

Personalia

Martyn Plummer, already a key contributor to the R project and in particular maintainer of the RedHat i386 GNU/Linux binary distribution and add-on package **coda**, has taken over as maintainer of the R GNOME interface.

Editors:
Kurt Hornik & Friedrich Leisch
Institut für Statistik und Wahrscheinlichkeitstheorie
Technische Universität Wien
Wiedner Hauptstraße 8-10/1071
A-1040 Wien, Austria

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 or Friedrich Leisch (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/`