Implementing a Class of Structural Change Tests: An Econometric Computing Approach

Achim Zeileis

Institut für Statistik und Mathematik, Wirtschaftsuniversität Wien, Austria

Abstract

The implementation of a recently suggested class of structural change tests, which test for parameter instability in general parametric models, in the R language for statistical computing is described: Focus is given to the question how the conceptual tools can be translated into computational tools that reflect the properties and flexibility of the underlying econometric methodology while being numerically reliable and easy to use.

More precisely, the class of generalized M-fluctuation tests is implemented in the package **strucchange** providing easily extensible functions for computing empirical fluctuation processes and automatic tabulation of critical values for a functional capturing excessive fluctuations. Traditional significance tests are supplemented by graphical methods which do not only visualize the result of the testing procedure but also convey information about the nature and timing of the structural change and which component of the parametric model is affected by it.

Keywords: econometric computing; R; structural change; M-estimation; fluctuation tests.

1 Introduction

This paper is about combining two fields of econometric research—testing for structural change and econometric computing—with the following focus: How can a general class of tests be translated into a unified and sound implementation that reflects the conceptual properties of the theoretical tests and allows for as much flexibility as possible in applications?

Structural change has been receiving attention in many fields of research and data analysis, in particular in time series econometrics: to learn if, when and how the structure of the data generating mechanism underlying a set of observations changes. Starting from the Recursive CUSUM test of Brown, Durbin, and Evans (1975) a large variety of fluctuation tests for structural change has been suggested and Kuan and Hornik (1995) introduced a generalized and unifying framework of fluctuation tests for linear regression models which has been extended to general parametric models estimated with M-type estimators by Zeileis and Hornik (2003). Among the special cases of this general class of tests are several well-known tests like the OLS-based CUSUM test (Ploberger and Krämer 1992, 1996), the Nyblom-Hansen test (Nyblom 1989; Hansen 1992), the class of tests of Hjort and Koning (2002) and the supLM, aveLM and expLM tests of Andrews (1993) and Andrews and Ploberger (1994).

The generalized M-fluctuation test class provides a conceptual tool box for constructing structural change tests in the following steps: 1. estimate the model which should be tested for structural instabilities, 2. derive an empirical fluctuation process from the cumulative sums of the M-estimation scores that is governed by a functional central limit theorem and that captures fluctuations over time, 3. aggregate the empirical fluctuation process to a scalar test statistic, augmented by a suitable visualization technique. The idea for step 3 is that boundaries can be chosen that are crossed by the limiting process (or some functional of it) only with a known probability α. Hence, if the empirical fluctuation process crosses these theoretical boundaries the fluctuation is improbably large and the null hypothesis of parameter stability can be rejected.

This is a preprint of an article published in *Computational Statistics & Data Analysis 50 (2006) 2987–3008* doi:10.1016/j.csda.2005.07.001

Computing and computational methods play an important role in econometric practice and (applied) research. However, there is a broad spectrum of various possibilities of combining econometrics and computing: two terms which are sometimes used to denote different ends of this spectrum are 1. *computational econometrics* which is mainly about methods that require substantial computations (e.g., bootstrap or Monte Carlo methods), and 2. *econometric computing* which is about translating econometric ideas into software. Of course, both approaches are closely related and cannot be clearly separated, but this paper is mostly concerned with *econometric computing* in the above sense and it is of interest beyond the structural change scope as it illustrates how a flexible and general class of tests can be translated into a set of modern computational tools. We discuss strategies for such an implementation based on language features including object orientation, functions as first class objects, nested lexically scoped functions and incorporation of re-usable components. These strategies are applied subsequently to the class of generalized M-fluctuation tests which are implemented in the package **strucchange** written in the R language for statistical computing (R Development Core Team 2005, see `http://www.R-project.org/`). R is an open-source system that offers all of the language features indicated above and is one of the most used environments for statistical computing—and although currently not the most popular language for econometric computing, R also finds increasing attention among econometricians (Cribari-Neto and Zarkos 1999; Racine and Hyndman 2002).

The remainder of this paper is organized as follows: Section 2 discusses the ideas and principles used for turning conceptual into computational tools independent of the programming language and in advance of the application to the problem of generalized M-fluctuation tests. Section 3 is concerned with the computation of M-fluctuation processes and Section 4 with the computation of the corresponding tests. In both sections, the underlying theory (established in Zeileis and Hornik 2003) is briefly reviewed before the translation into the R language is described. Section 5 illustrates the usage of the software in the analysis of artificial and real-world data before a brief summary and some conclusions are given in Section 6.

2 From conceptual to computational tools

2.1 Goals

What are desirable features of the implementation of an econometric procedure? It should be easy to use, numerically reliable, computationally efficient, flexible and extensible and reflect the features of the underlying conceptual method. In many situations, although not necessarily in all, some of these desired features might be antagonistic: e.g., some computational efficiency can usually be gained by sacrificing extensibility to a certain extent. Also, what is easy to use might be very different for different types of users: here, we assume that the users are at least comfortable with using a command line interface (CLI). Certainly, there are users that prefer simple graphical user interfaces (GUIs), but in a scientific context this approach is typically too limited to reflect the flexibility of a conceptual procedure. Furthermore, it is always possible to build a GUI on top of some CLI application, if it should be necessary to communicate some more specific functionality to a non-advanced group of users.

With the different goals stated above, what should be the guiding principle when implementing a new procedure? We advocate that the implementation should always be guided by the properties of the underlying procedure (while trying to ensure as much efficiency and accuracy as possible), i.e., the resulting functions should do that what we think the method does conceptually. This typically makes the functions easy to use for those who know the underlying methodology and, vice versa, theoretical concepts can also be communicated by supplying software that is easily applicable. And if the underlying method is flexible, such software often also becomes extensible and flexible in a rather natural fashion. Just implementing how we perceive a certain method and think about it may seem like a trivial principle, but many programs written today are still not guided by the underlying theory but rather by the limitations that programming languages used to have (and some still have), where programmers were forced to represent everything in

numeric vectors and matrices only.[1] Therefore, we will discuss briefly some language features that are available in modern programming languages and how these might be useful to accomplish our goals.[2]

2.2 Language features

Object orientation The advantages of object-orientated programming for statistical and econometric computing are probably well-known and it is rather obvious how object orientation can help to achieve the goals formulated in the previous paragraph: We can define classes whose instances can be rather complex objects that represent an abstraction of a certain procedure or type of data. And for these classes, methods performing typical tasks can be implemented (e.g., see Chambers and Hastie 1992; Doornik 2002).

Functions as first-class objects This concept is not implemented in many statistical/econometric computing environments and is thus not as well known as object orientation. It refers to functions being a basic data type that can be passed as a formal argument to another function and that can also be (part of) a return value of a function. To some degree this feature can be substituted by object orientation: for example, if we want to write a function that carries out computations based on the residuals of a fitted model (of arbitrary type), we could either assume the existence of a method for extracting residuals (thus, rely on object orientation) or we could expect the residual-extracting function as a second argument. Of course, both approaches can also be combined: the formal method could be used unless some other function is specified. The latter two approaches are more flexible—e.g., as there might be different kinds of residuals that could be of interest and are not all supported by the formal method—and require less discipline from users and developers when adding new functionality. Substituting functions as return values by means of object orientation is more complicated (and less intuitive) as we will emphasize in the next paragraph.

Lexical scope Lexical scoping—which is used here as a sloppy term for *nested lexically scoped functions*—is only an issue if functions can create other functions as their return value: If these returned functions have free variables (i.e., neither formal parameters nor local variables), the scoping rules determine where the values for these symbols are searched. With lexical scoping the values for free variables are stored in the so-called function closure, i.e., in the environment in which the function was created. This avoids carrying out computations several times in different places, yielding simpler and more intelligible implementations. Lexical scope has many applications in statistics (see Gentleman and Ihaka 2000), e.g., for computing distributions, bootstrapping or optimization. A simple example would be the computation of a maximum likelihood estimate. In a first step, the likelihood function could be set up by taking the data as input and returning a function that only depends on the parameters. Thus, the data would not have to be passed around explicitly and we could focus on the object of interest: the likelihood as a function of the parameters. In a second step, this function could be passed on to the optimizer which can return the estimated parameters. See Gentleman and Ihaka (2000) for a detailed discussion.

Compiled code For most programmers it is easier and more convenient to write in interpreted languages, but efficiency can be gained if compiled code is used. To enjoy the advantages of both, there are different approaches: some interpreted languages allow for (byte) compilation, others allow for dynamic linking of compiled code such that the more convenient user interface in interpreted code is retained while doing the number crunching in compiled code. When implementing a

[1] Of course, every implementation is limited by the features of the language it is written in, but there are certainly more flexible tools available than exploited by many programmers of econometric functions.

[2] This discussion should serve as a motivation and is thus rather non-technical and in some places not very precise from a computer-scientific perspective.

new procedure, developers should be aware of both possibilities and try to combine computational efficiency with a user-friendly interface.

Re-usable components This is not strictly a programming language feature in a computer-scientific sense, but important for the choice of the environment in which a new procedure should be implemented and hence discussed here. Of course, it is desirable that the environment chosen provides components that new functions can build on—and that encourages programmers to implement procedures that can again be used as building blocks by others. Thus, ideally we want to use a system that already provides us with many tools, and it should be used in such a way that existing functionality is not re-implemented. Similarly, the new programs should also be written in such a fashion that the functionality is easily accessible to other developers which can in turn re-use these new building blocks. For example, for doing computations in a standard model, e.g., linear regression, a numerically reliable OLS fitter for linear models should be provided by the system (and not be re-implemented by the programmer), whereas for computations in non-standard models, the new fitter for such a model should be provided by the developer as a stand-alone function (for other developers to use) and further computations should only be done subsequently (and not both steps in a single function).

To sum up, these features could be combined in a strategy for implementing an econometric procedure: First, it needs to be clear what the important properties of the conceptual method are that is going to be implemented. Then, an abstraction in the form of a function, an object, or a class structure reflecting these properties can be created. If possible, this should build on existing functions and classes, and if necessary, computationally intensive parts can be placed into more efficient compiled code. Employing a functional approach for this, is typically more intuitive as it corresponds to the way we deal with many methods analytically. Hence, where appropriate, procedures should take functions as arguments and return functions (which might be simplified if nested lexically scoped functions are available).

Although none of the features discussed above are really new, and have been available in some form in statistical computing environments for about 10-15 years, many programmers still use languages that provide very little support in these directions or they use the languages as if they did not provide these features: i.e., what is implemented more often than not are single monolithic functions that return a lot of potentially useful output given some starting information. This might be considered to be 'easy to use' if one is only interested in this particular procedure, but such programs are usually hard to extend and whether or not they are reliable is also often by no means obvious.

2.3 A simple example

With respect to the main topic of this paper "implementing a class of structural change tests", the principles discussed are still very vague. The reason is that it is difficult to give general rules or guidelines for the implementation of tests because they depend so heavily on the challenges the particular class of tests poses. Nevertheless, before going into details about how this problem is solved for the specific class of generalized M-fluctuation tests, some aspects should be motivated that typically occur when implementing a model-based test with a non-standard distribution.

Let us assume that there are different types of models from which we can extract some quantity of interest (e.g., residuals, scores or coefficients) based on which a test statistic should be derived. Furthermore, the (asymptotic) distribution is not known, non-standard or does depend on the data, such that it has to be approximated by some sort of simulation. To translate this into a function `modeltest`, in the first step the fitted model needs to be computed: we can either expect the user to have done that, or do it ourselves by asking the user for a model specification plus a model fitter. Or we can support both and interpret the corresponding argument as a fitted model if no fitting function is supplied and regard it as a model specification if a fitter is specified. In a second step, an extractor function is needed that computes the residuals, say, for which a method to a generic function is available. Therefore, the arguments of the function `modeltest` could be

```
modeltest(obj, fit = NULL, extractor = residuals, n = 10000)
```

where `obj` has to be either a fitted model (when `fit = NULL`) or a model specification to be fitted by `fit`. The `extractor` function is by default `residuals` but could be set to some other function. The argument `n` should be the number of simulated replications for the distribution of the test statistic. The body of the function would then have

```
obj <- if(!is.null(fit)) fit(obj)

statistic <- compute_statistic(extractor(obj))
sim_stat <- simulate_statistic(n, extractor(obj))

p_fun <- function(x) sum(sim_stat > x)/n
q_fun <- function(x) quantile(sim_stat, x)

return(list(fitted_model = obj,
            statistic = statistic,
            p_value = p_fun(statistic),
            p_fun = p_fun,
            q_fun = q_fun))
```

First, the model is fitted (if necessary), then the test statistic is computed based on the quantity of interest, and `n` simulated replications from the distribution of the statistic are generated under the null hypothesis. Here, we assume that `compute_statistic` is a known function and `simulate_statistic` is an algorithm like bootstrap, MCMC, permutation or simulation from an asymptotic distribution. Based on the `n` replications `sim_stat` the empirical p value function `p_fun` and the empirical quantile function `q_fun` can be defined. Finally, model, statistic, p value, p value function and quantile function can be returned. The latter two are defined using lexical scoping, the free variables `sim_stat` and `n` are not exported and only stored in the closure of the functions. So the question could be asked why we should return these at all or why not simply return `sim_stat` instead? The simulated distribution can be useful for other tasks, e.g., visualizing critical values (at user-defined significance levels not known in advance) so it should be stored in some way; and as we are typically interested in the distribution only for computing critical values and p values, this interface is much more convenient and the `p_fun` and `q_fun` should not have to be set up again every time another method is applied.

Although this example is very simple, it clarifies some of the basic ideas for the implementation expounded in the following. The class of M-fluctuation tests poses a few further challenges, not yet addressed: in particular, the function called `compute_statistic` above on which all further steps like simulation or visualization depend can also be user-defined. For the implementation we exploit all language features discussed in the previous section except compiled code because all computations carried out are fairly simple. The bottleneck for M-fluctuation tests is typically the model-fitting and not the computation of the test statistic. Our implementation is object-oriented and relies on nested lexically scoped functions and builds on/provides re-usable components. The only comprehensive system offering statistical functionality and high quality computer graphics that incorporates all of the required features is the R system for statistical computing. Other object-oriented systems such as Ox or MATLAB, which are probably more popular for econometric tasks, could also be used but certain tasks would have to solved differently than described here. In languages without object orientation, such as GAUSS, it would be very hard to derive an implementation with the same generality as our suggested solution.

2.4 Software delivery

Another advantage of using R is that software delivery—a very important issue when new software is implemented—is simple: in our case, the generalized M-fluctuation tests are implemented in

the package **strucchange** (Zeileis, Leisch, Hornik, and Kleiber 2002; Zeileis, Kleiber, Krämer, and Hornik 2003)—mainly in the new functions `gefp` and `efpFunctional`—which is, like R itself, available at no cost under the terms of the general public licence (GPL) from the comprehensive R archive network at http://CRAN.R-project.org/. Therefore, the software can not only be used by everyone interested, the whole source code can be inspected. Hence, with the wording from Claerbout's principle (see Leisch and Rossini 2003, for a discussion), this article is not only an advertisement for a scholarship (in form of the **strucchange** package), the scholarship itself is also freely available. This assures easy reproducibility of the results in this paper; an issue which received increased attention in both statistics and econometrics (McCullough and Vinod 2003).

3 Generalized M-fluctuation processes

3.1 Theory

In structural change problems, it is of interest to test the hypothesis that the parameters of a certain model remain constant over all observations against the alternative that they change over "time".

More formally, we have a parametric model with k-dimensional parameter θ_i for n ordered observations of a possibly vector-valued variable Y_i $(i = 1, \ldots, n)$. Under the assumption that for each observation at time i there is a parameter θ_i such that the model holds, the null hypothesis is that the parameters are constant over the full sample period

$$H_0 : \quad \theta_i = \theta_0 \qquad (i = 1, \ldots, n)$$

which is tested against the alternative that θ_i changes over "time". The tests described in the following have power against various patterns of departures from parameter constancy. In time series applications, the observations are typically ordered with respect to time, but different orderings may be more natural in other applications: e.g., ordering with respect to a specific prognostic factor in biometric studies or with respect to income in economic models.

The idea of the tests is to estimate the parameter vector once for all n observations based on an M-estimation score function $\psi(\cdot)$ which has zero expectation at the true parameters $\mathsf{E}(\psi(Y_i, \theta_i)) = 0$ and to use the (scaled) cumulative sum process of these scores to detect instabilities in the parameters. The full sample M-estimator is implicitly defined by

$$\sum_{i=1}^{n} \psi(Y_i, \hat{\theta}) \quad = \quad 0. \tag{1}$$

This includes various estimation techniques as special cases, such as ordinary least squares (OLS), maximum likelihood (ML), instrumental variables (IV), (generalized) estimating equations (GEE), robust M-estimation, Quasi-ML. Generalized method of moments (GMM) is also closely related to this class of estimators.

Under parameter stability, the scores $\psi(Y_i, \hat{\theta})$ have zero mean but under the alternative there will be systematic deviations from zero which can be captured using the cumulative sum process of the scores:

$$W_n(t) \quad = \quad n^{-1/2} \sum_{i=1}^{\lfloor nt \rfloor} \psi(Y_i, \hat{\theta}). \tag{2}$$

It can be shown that for this process a functional central limit theorem (FCLT) holds which implies that the process $W(\cdot)$ converges to a process $Z(\cdot)$ that is a Gaussian process with zero mean and covariance function $\mathsf{COV}[Z(t), Z(s)] = \min(t, s) \cdot J$, where J is the (asymptotic) covariance matrix of the scores ψ. Usually, this covariance structure can easily be estimated, the simplest estimator being

$$\hat{J} \quad = \quad n^{-1} \sum_{i=1}^{n} \psi(Y_i, \hat{\theta}) \psi(Y_i, \hat{\theta})^\top, \tag{3}$$

but under heteroskedasticity or serial correlation more elaborate estimators that are consistent under weaker assumptions should be used, e.g., HC (White 1980; Cribari-Neto 2004) or HAC type estimators (Andrews 1991; Lumley and Heagerty 1999). Given such a consistent covariance matrix estimate $\hat{J}$ the decorrelated empirical fluctuation process $efp(\cdot)$ can be computed

$$efp(t) \quad = \quad \hat{J}^{-1/2} W_n(t), \tag{4}$$

which converges to a Brownian bridge $W^0(\cdot)$

$$efp(\cdot) \quad \xrightarrow{\text{d}} \quad W^0(\cdot). \tag{5}$$

This FCLT provides the probabilistic basis for the structural change tests in Section 4 but before that the implementation of the generalized M-fluctuation processes $efp(\cdot)$ is discussed.

3.2 Implementation

For computing an empirical fluctuation process $efp(\cdot)$ defined in Equation (4) the estimate of the model parameters $\hat{\theta}$ is required. In principle, it would be sufficient to specify the data Y_i and the score function $\psi(\cdot)$ for which Equation (1) could be solved by some generic optimizer. But in practice, this could lead to severe numerical problems as different functional forms of $\psi(\cdot)$ usually require different optimization techniques. Fortunately, we do not need to re-invent the wheel here as we can use existing well-established R functions to fit many popular models and we just need to extract the scores or estimating functions at the fitted model. This has two advantages: 1. the usual model specification interface can be used for fitting a model and 2. the numerical accuracy and computational efficiency of the model fitting functions already available in R can be exploited.

Thus, we just need a model fitter with corresponding model specification and a function to extract the scores $\psi(Y_i, \hat{\theta})$ and then we can provide infrastructure to compute the empirical fluctuation process $efp(\cdot)$ based on these. Furthermore, it would be desirable to plug in a covariance matrix estimator for $\hat{J}$ which is also evaluated at the fitted model so that neither the model/scores nor the covariance matrix estimate has to be computed by the user beforehand.

These ideas have been implemented in the function `gefp` in the package **strucchange**. The most important arguments of `gefp` are:

```
gefp(..., fit = glm, scores = estfun, vcov = NULL, data = list())
```

where `...` can be an arbitrary model specification which is passed together with the `data` (a `data.frame` that holds the data and is by default empty) to the model fitting function `fit`—by default `glm` for fitting generalized linear models (GLMs). If `fit` is set to `NULL`, `...` is assumed to be already the fitted model. The argument `scores` is the function responsible for extracting the matrix of scores $(\psi(Y_i, \hat{\theta}))$ from the fitted model—by default this is the generic function `estfun` which has methods for extracting the estimating functions (or scores) from linear models (fitted by `lm`), GLMs including logit and probit models (fitted by `glm`) and robust regressions (fitted by `rlm` in **MASS**). The argument `vcov` is a function for computing some covariance matrix estimate $\hat{J}$—by default it is defined to compute the estimate defined in Equation (3).

The essential steps carried out by `gefp` in computing $efp(\cdot)$ are:

```
fm <- if(is.null(fit)) list(...)[[1]]
        else fit(..., data = data)
psi <- scores(fm)
J <- vcov(fm)
n <- nrow(psi)
process <- apply(psi, 2, cumsum)/sqrt(n)
```

Thus, in a first step the model fitting function is applied (if necessary) to the model specification and the data, yielding a fitted model `fm`. From this the scores `psi` and the covariance matrix estimate `J` are extracted before the cumulative sum process `process` from Equation (2) is computed.

Note, that the arguments `fit`, `scores` and `vcov` are all full functions whose evaluation is taken on by `gefp` and does not have to be done by the user.

Of course, the actual function `gefp` is much longer and provides several additional features, but the code above contains the core steps which have been just slightly simplified to illustrate the procedure. The function `gefp` then returns an object of class `"gefp"` which contains in particular the empirical fluctuation process along with some meta-information such as the fitted model. The full list of arguments is

```
gefp(..., fit = glm, scores = estfun, vcov = NULL, data = list(),
     fitArgs = NULL, order.by = NULL,
     sandwich = TRUE, parm = NULL, decorrelate = TRUE)
```

The arguments in the first line have already been discussed above. The arguments in the second line provide a way to specify further arguments to `fit` via `fitArgs` and `order.by` can be used to define an ordering, e.g., based on a regressor, while the default is that the observations are already ordered. The third line gives a few arguments that allow for finer control of the computation of the empirical fluctuation process: `sandwich` specifies whether the matrix computed by `vcov` is the full sandwich estimate or only the "meat" of the sandwich; `parms` can be set to a subset of all parameters estimated, corresponding to selecting only a few columns of the fluctuation process and `decorrelate` specifies whether the process should be decorrelated by multiplication with $\hat{J}^{-1/2}$ or whether the process should only be scaled using the diagonal elements of $\hat{J}^{-1/2}$. The latter is useful if only a single CUSUM-type process of "residuals" should be computed.

By now, we can compute a fitted object representing an empirical fluctuation process for which the limiting process is known. This is the basic probabilistic ingredient for computing the limiting distribution for the significance tests discussed in the next section.

4 Generalized M-fluctuation tests

4.1 Theory

In the previous section, we derived an empirical fluctuation process $efp(\cdot)$ (see Equation (4)) that is able to capture instabilities in the parameter estimates of a model. It is already possible to visually inspect this process for excessive fluctuations but for inference we want to aggregate it to a test statistic using some scalar functional $\lambda(efp)$ that brings out deviations from the limiting process.

What are sensible choices for $\lambda(\cdot)$? In finite samples, the process is essentially a matrix $(efp_j(i/n))_{i,j}$ with $i = 1, \ldots, n$ corresponding to "time" points and $j = 1, \ldots, k$ corresponding to the independent components of the process which are typically components of the parameter vector θ. Hence, two strategies for constructing a test statistic are straightforward: 1. We can aggregate the process over time first, yielding k independent test statistics each associated with one component of the process. 2. We can first aggregate over the components yielding a fluctuation process which can reveal the timing of a potential structural change. More formally, this means that in most situations we can split λ into two parts: λ_{time} and λ_{comp} for aggregation over time and components respectively. Typical choices for λ_{time} are the absolute maximum L_∞, the mean or range—and for λ_{comp} again L_∞ or the squared Euclidean norm L_2 (see Hjort and Koning 2002, for more examples). Depending on the order of the aggregation, as pointed out above, either the timing of the shift or the component affected by it can be identified.

The only functional which allows for both interpretations is the double maximum statistic

$$\max_{i=1,\ldots,n} \max_{j=1,\ldots,k} \left| \frac{efp_j(i/n)}{b(i/n)} \right|, \tag{6}$$

where L_∞ is used for aggregation in both directions and the null hypothesis of stability is rejected when the maximum exceeds some boundary function $b(t)$ which is typically chosen to be constant, i.e., $b(t) = 1$.

Another popular statistic (e.g., used by Nyblom 1989; Hansen 1992, among others) is the Cramér-von Mises statistic which has some optimality properties for random walk alternatives. It can be constructed by aggregating first over the components using the squared Euclidean norm L_2 and then over time by taking the mean.

$$n^{-1} \sum_{i=1}^{n} ||efp(i/n)||_2^2 . \tag{7}$$

In situations where there is shift in the parameters and then a second shift back, it is advantageous to aggregate over time using the range and reject if the maximum range becomes too large. The corresponding test statistic is

$$\max_{j=1,\ldots,k} \text{range}_{i=1,\ldots,n} efp_j(i/n). \tag{8}$$

Critical values for arbitrary functionals can easily be obtained by simulation of $\lambda(W^0)$: justified by the FCLT from Equation (5) we can simulate Brownian bridges and apply the functional $\lambda(\cdot)$ to them to obtain a sample from the limiting distribution of the test statistic. In certain special cases, closed form solutions are also known: series expansions are available for functional (6) with constant boundary (Ploberger and Krämer 1992) and functional (8) (**?**). For the functional (7), simulated asymptotic critical values can be found in Hansen (1992).

4.2 Implementation

Given the flexibility of the generalized M-fluctuation test framework described in the previous section, we would like to have an implementation that reflects this flexibility and provides support for the full class of tests and not only certain special cases. Therefore, we want to translate the simple conceptual idea of applying an arbitrary functional λ to an empirical fluctuation process *efp* into an equally simple computational tool where we just need to specify the functional and can then apply it to `"gefp"` objects containing the empirical fluctuation process. The implementation idea is that based on a specification of the functional critical values are simulated (or a closed form solution is used if supplied), a sensible visualization technique is chosen which reflects the properties of the test statistic and then an integrated object is returned which knows about process visualization, computation of the test statistic and the corresponding p values. Thus, this object provides infrastructure which can be used by methods for the generic functions `plot` for visualization and `sctest` (structural change test) for significance testing.

Of course, the simulation of critical values could in principle also be carried out on the fly, but as many rather high-dimensional processes might have to be generated for obtaining reliable p values this can be burdensome and it makes the usage much easier if p values are just simulated once and stored in a suitable object. At first sight, this does not seem very different from the traditional approach of producing printed tables of critical values, but it has two advantages: 1. Producing and printing tables as well as looking at them is tedious. Users are usually not really interested in the table itself but only in the computation of a critical value, or preferably a p value, and it is much more convenient if the computer handles the look-up. 2. It gives the user much more freedom and flexibility in trying out new functionals by quickly simulating critical values based on a smaller number of replications which gives sufficiently good results for exploration. When more reliable p values are needed, production quality tables can be computed by more extensive simulations with a higher number of replications and stored again in a convenient integrated object.

For several frequently used functionals such objects are readily available in **strucchange** including `maxBB` for the double maximum functional from Equation (6), `meanL2BB` for the Cramér-von Mises functional from Equation (7) and `rangeBB` for the range test from Equation (8).

New objects for user-specified functionals can be easily generated with the function `efpFunctional` whose most important arguments are

```
efpFunctional(
  functional = list(comp = function(x) max(abs(x)), time = max),
  boundary = function(x) rep(1, length(x)),
  computePval = NULL, computeCritval = NULL,
  nobs = 10000, nrep = 50000, nproc = 1:20)
```

Obviously, the most important argument is `functional` which can either be a single function or a list of two functions named `comp` and `time` to be applied in the order of appearance. Hence, the default is to first aggregate over the components using the absolute maximum and then over time using the maximum. The `boundary` argument is used for weighting of the process after the first of the two parts of the functional has been applied, the default is a constant boundary. If either of the arguments `computePval` or `computeCritval` is specified, they are used for computing the limiting distribution, otherwise it is simulated using the settings in `nobs`, `nrep` and `nproc`.

The function `computePval` has to take two arguments, the test statistic and the number of processes k, and has to return the corresponding p value. `computeCritval` is the inverse function, i.e., given a significance level and the number of processes k it should return the corresponding critical value. If neither of those functions is specified (i.e., both set to `NULL`) `nrep` replications of `nproc`-dimensional Brownian bridges, each consisting of `nobs` observations, are generated and subsequently the specified functional is applied to obtain a sample from $\lambda(W^0)$. The empirical quantile function evaluated at a grid of p values is then used for computing a table of critical values. By default, the p values are in 0.1% steps up to 15% and in 1% steps up to 100% (but could be changed using the argument `probs` not listed above). If `nproc` is set to `NULL` only a 1-dimensional Brownian bridge is simulated and p values for higher-dimensional processes are derived from a Bonferroni correction: i.e., if p is the 1-dimensional p value the k-dimensional p value is $1-(1-p)^k$. This is appropriate and saves a lot of computation time if the independent component statistics are aggregated using the maximum.

An object created by `efpFunctional` is of class `"efpFunctional"` and has slots with functions `plotProcess`, `computeStatistic` and `computePval` that are defined based on lexical scoping. That is, although basically the usual table of significance levels and critical values has been tabulated internally, the user is provided with a much more convenient interface for performing significance tests. In a language without nested lexically scoped functions, a workaround would be to return the classical table of critical values and to set up the functions separately again in each of the methods applied subsequently.

The `plotProcess` function chooses a suitable visualization technique for the particular functional specified. Either it produces a time series plot if λ_{comp} is applied first or it produces a histogram-like line plot if λ_{time} is applied first. In both cases, the boundary representing the critical value is added and if a functional other than the maximum is applied second a dashed line representing the test statistic is drawn. This allows for carrying out the significance test graphically and at the same time conveying information about the timing and/or the component of the shift. If different visualization techniques should be applied a suitable function `plotProcess` can be supplied in the call to `efpFunctional`.

The various options in `efpFunctional` provide the fine control required for incorporating the heterogeneous possibilities covered by the underlying M-fluctuation test theory. But in most situations, the application of the functions is very easy at the user-level: first, an empirical fluctuation process object `gefp.obj` for a certain model is fitted by `gefp` and second a functional object `efpFun.obj` can be computed by `efpFunctional` (if not already available in **strucchange**). Subsequently, significance tests can be carried out by visualization or in the traditional way by computing a test statistic and a p value. In R, the commands are simply

```
plot(gefp.obj, functional = efpFun.obj)
sctest(gefp.obj, functional = efpFun.obj)
```

using the methods for the generic functions `plot` and `sctest`. Examples based on artificial and real-word data are provided in the next section.

5 Applications and illustrations

In this section, we apply the previously established conceptual and computational tools to two real-world data sets which exhibit structural changes. In both cases, GLMs are fitted to the data, the corresponding ML scores are used for constructing the empirical fluctuation processes and several combinations of λ_{comp} and λ_{time} functionals are used for visualizing the corresponding significance tests. In addition, two artificial data sets are used to illustrate how non-standard regression techniques can easily be plugged into the functions discussed. More specifically, generalized M-fluctuation tests will be used for detecting parameter instabilities in beta regression (Ferrari and Cribari-Neto 2004).

The R commands presented are sufficient for reproducing the results.

5.1 Boston homicides

The "Boston Gun Project" was launched in the mid-1990s to address the problem of high youth homicide rates in Boston. This policy intervention implemented the "Operation Ceasefire" which aimed at lowering the homicide rate by a deterrence strategy. As it is unknown when this intervention became effective Piehl, Cooper, Braga, and Kennedy (2003) use modified versions of the sup$F$ test (Andrews 1993) to assess whether the project was a success.

Here, we use a simple Poisson GLM for the mean number of youth homicides per month, the corresponding time series is depicted in the left panel of Figure 1. As the additional regressors population and seasonality used by Piehl *et al.* (2003) in their model A have no significant influence at a 5% level we just fit an intercept to the data. The null hypothesis of a constant mean over the full sample period from 1992(1) to 1998(5) is tested with the M-fluctuation test based on the ML scores of the GLM and a quadratic spectral kernel HAC estimator (Andrews 1991) with VAR(1) prewhitening and automatic bandwidth selection based on an AR(1) approximation for estimating the covariance matrix $\hat{J}$.

The package **strucchange** and the Boston homicide data contained in the package are loaded via

```
R> library(strucchange)
R> data(BostonHomicide)
```

Then, we can fit the empirical fluctuation process using the function `gefp`:

```
R> bh.efp <- gefp(homicides ~ 1, family = poisson, data = BostonHomicide,
+     vcov = kernHAC)
```

The formula `homicides ~ 1` and the `family = poisson` arguments are passed together with the data frame `BostonHomicide` to the default fitting function `glm` which fits a Poisson GLM for the intercept of the homicides time series. The `vcov` argument is set to the function `kernHAC` which is contained in the package **sandwich** (Zeileis 2004b), that is required by **strucchange** and implements the kernel HAC estimator.

For assessing the significance of any changes in the mean number of youth homicides the Cramér-von Mises statistic from Equation (7) is used which is already available as the `"efpFunctional"` object `meanL2BB` in **strucchange**. The significance test can be carried out by

```
R> sctest(bh.efp, functional = meanL2BB)

        M-fluctuation test

data:  bh.efp
f(efp) = 0.9337, p-value = 0.005
```

yielding a test statistic of 0.934 with a highly significant approximate asymptotic p value of 0.005 (the lowest p value stored in `meanL2BB`). A more informative way of performing the same test is the visualization produced by

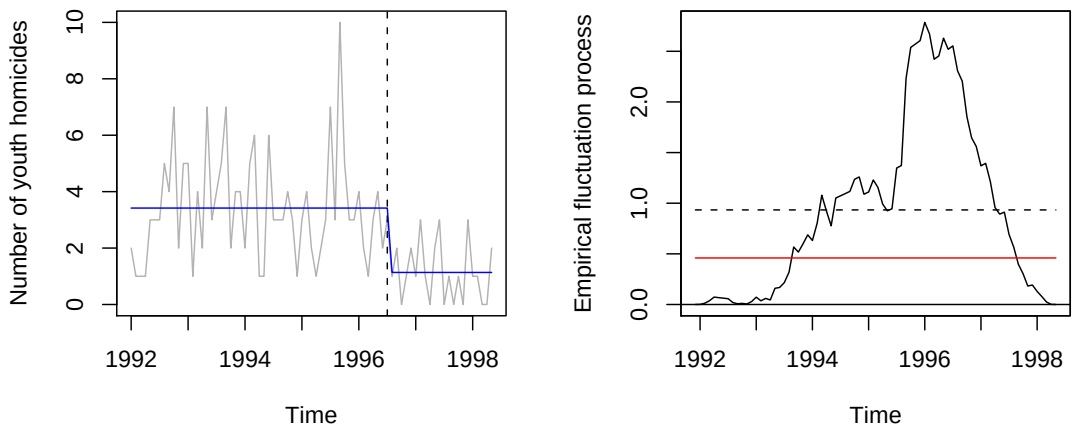


Figure 1: Boston homicides (left) and empirical M-fluctuation process with mean L_2 norm (right).

```
R> plot(bh.efp, functional = meanL2BB)
```

The resulting display can be found in the right panel of Figure 1 which conveys two messages: 1. As the test statistic (visualized by the dashed horizontal line) exceeds its default 5% critical value (solid horizontal line) there is evidence for a change in the mean number of youth homicides. 2. The peak in early 1996 suggests that there has been a single shift in the mean at around that time. The corresponding segmented regression model with an estimated changepoint in 1996(7) (estimated by maximizing the segmented likelihood) is depicted in the left panel of Figure 1.

5.2 Austrian national guest survey

To illustrate the usage of the M-fluctuation test framework in other scenarios than time series regression, we analyze data provided by the Austrian national tourist office from the Austrian national guest survey about the summer seasons 1994 and 1997. Aggregated data from this survey are available from TourMIS (http://tourmis.wu-wien.ac.at/), an online Marketing-Information-System for tourism managers, a subset of the disaggregated data are analyzed in Dolnicar and Leisch (2000). The subset of the data used in this paper can be obtained from http://www.ci.tuwien.ac.at/~zeileis/data/gsa.rda and then loaded by `load("gsa.rda")`. In the following, we investigate how the preference for cycling as a summer vacation activity varies with the available socio-economic indicators using a logistic regression model based on the variables `Cycle` (cycling done/not done), `Age` (age in years), `HHIncome` (household income in ATS per month), `Gender` (male/female) and `Year` (1994/1997). The gender has only a slight but significant influence on the preference for cycling so for simplicity the model is fitted just for the subset of 6256 male tourists which does not substantially change the results of the parameter instability tests presented below. The year has a significant influence—cycling became on average slightly more popular from 1994 to 1997—and a polynomial of degree 2 in age is appropriate—cycling becomes increasingly less popular for older tourists. However, the household income is not significant and hence was not included in the logistic regression model. As changes in the preference for cycling with income would seem intuitive, we test the structural stability of the coefficients of the logistic regression model over log-income.[3]

[3] Of course, ordering by income would yield equivalent results, but the graphics are much more intelligible if a

In R, logistic regression models are fitted as binomial GLMs, again via the function `glm`. Therefore, the empirical M-fluctuation process can be constructed from the ML scores of the model `Cycle ~ poly(Age, 2) + Year` using the following call to `gefp`:

```
R> gsa.efp <- gefp(Cycle ~ poly(Age, 2) + Year, family = binomial,
+     data = gsa, order.by = ~log(HHIncome), parm = 1:3)
```

The `order.by` argument specifies that changes over increasing (log-)income should be detected and `parm = 1:3` specifies that only the first three parameters (intercept and the two coefficients of the age polynomial) are tested, i.e., we do not test for changes in the year effect as interaction between income and year seems less plausible.

To assess the parameter stability in this model several versions of the double maximum statistic from Equation (6) are used in order to illustrate the usage of `efpFunctional`. First, we use the version with a constant boundary already available in the `maxBB` object which first aggregates over the components and then over time. The command `plot(gsa.efp, functional = maxBB)` produces a plot of the aggregated process which crosses its 5% level boundary—hence, there are significant parameter instabilities in the model—and shows two peaks at about the lower quartile 9.99 and upper quartile 10.65 of log-income which can be interpreted as the "timing" of the shifts.

To additionally reveal the component of the shift we look at the non-aggregated process: as pointed out in Section 4 the double maximum tests are the only which allow for both identification of timing and component of structural changes which means that in the visualization also boundaries for the non-aggregated process can be plotted. Such a display for the same functional is produced by

```
R> plot(gsa.efp, functional = maxBB, aggregate = FALSE)
```

whose results are shown in Figure 2. It shows that both the intercept and the quadratic term in the age polynomial exhibit structural instabilities over income as the corresponding processes cross their boundaries with peaks at the "times" described above. The first shift affects the quadratic age term and the second shift the intercept; the linear term shows some fluctuations at both points but none of which are significant. These instabilities show that the income has a significant influence on the preference for cycling and that there is also an interaction effect with age. However, this influence is not linear—otherwise this would have been picked up by including the variable `log(HHIncome)` into the regression model—but somewhat step-shaped. The reason is that for tourists with a low income (below the lower quartile) cycling is much more popular with young tourists and less popular with middle-aged tourists compared to the higher income groups.

The corresponding formal significance test can be carried out by calling `sctest(gsa.efp, functional = maxBB)` which gives a test statistic of 2.059 and a highly significant asymptotic p value of 0.001.

To illustrate the usage of `efpFunctional`, we consider two other versions of double maximum tests with a different aggregation strategy (`dmax1`) and a different boundary function (`dmax2`) . The functional `dmax1` created by

```
R> dmax1 <- efpFunctional(functional = list(time = function(x) max(abs(x)),
+     comp = max), computePval = maxBB$computePval)
```

produces an equivalent test with the same test statistic as `maxBB`, hence the same closed-form p value function can be used. The only difference is the order of aggregation: `dmax1` first aggregates over time and then over the components which leads to a different visualization: the output of `plot(gsa.efp, functional = dmax1)` is depicted in the left panel of Figure 3. It gives the same results as in Figure 2—the coefficients for the intercept and the quadratic term in the age polynomial are responsible for the shift—it just provides another (and for this particular combination of functionals less informative) view on the same process. The output of `sctest(gsa.efp, functional = dmax1)` is identical to that using `maxBB` described above. If a functional should

log-scale is used.

be applied which leads to a similar visualization but where the order of aggregation cannot be interchanged, the range test via `plot(gsa.efp, functional = rangeBB)` could be used leading to a very similar visualization.

The functional `dmax2` uses the same functional as `maxBB` but with a different boundary function which is proportional to the standard deviation of the Brownian bridge plus a small intercept $b(t) = t \cdot (1-t) + 0.05$ (similar to the functional considered in Zeileis 2004a). The functional `dmax2` is created by

```
R> set.seed(123)
R> dmax2 <- efpFunctional(functional = list(comp = function(x) max(abs(x)),
+     time = max), boundary = function(x) sqrt(x * (1 - x)) + 0.05,
+     nobs = 1000, nrep = 1000, nproc = NULL)
```

which simulates a table of critical values based on a Bonferroni approximation. The seed of the random number generator and the low number of observations and replications in the simulation are chosen for making the results quickly reproducible for the reader; they do not introduce much bias in the p values and are sufficient for illustration purposes. The process, aggregated over components, together with its new boundary can be easily displayed by `plot(gsa.efp, functional = dmax2)` leading to the plot in the right panel of Figure 3. The corresponding structural change test, again carried out by `sctest(gsa.efp, functional = dmax2)`, gives the test statistic 4.795 and an approximate asymptotic p value of 0 (which just reflects that none of the `nrep = 1000` simulated Brownian bridges led to a greater test statistic).

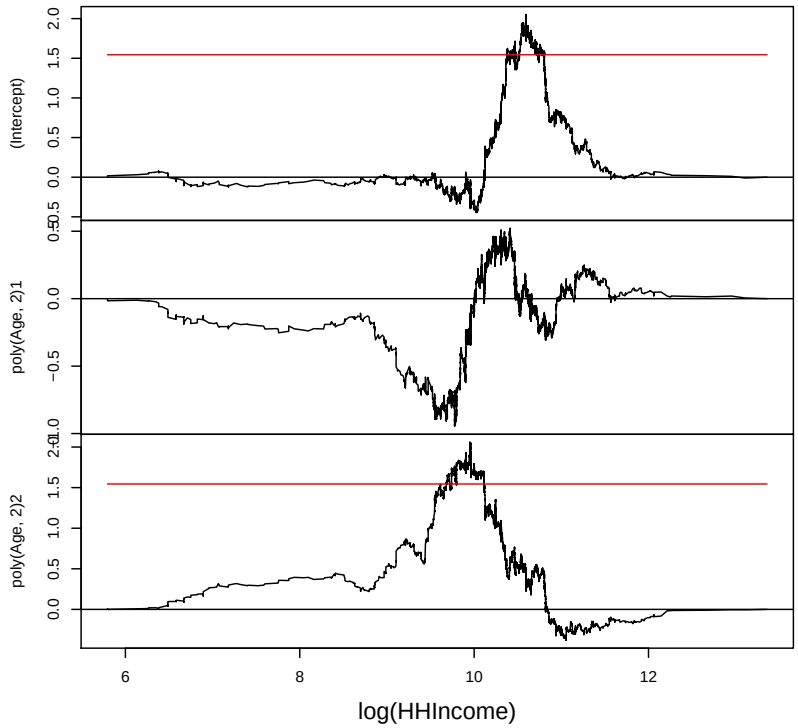


Figure 2: 3-dimensional empirical M-fluctuation process for guest survey data with double maximum functional.

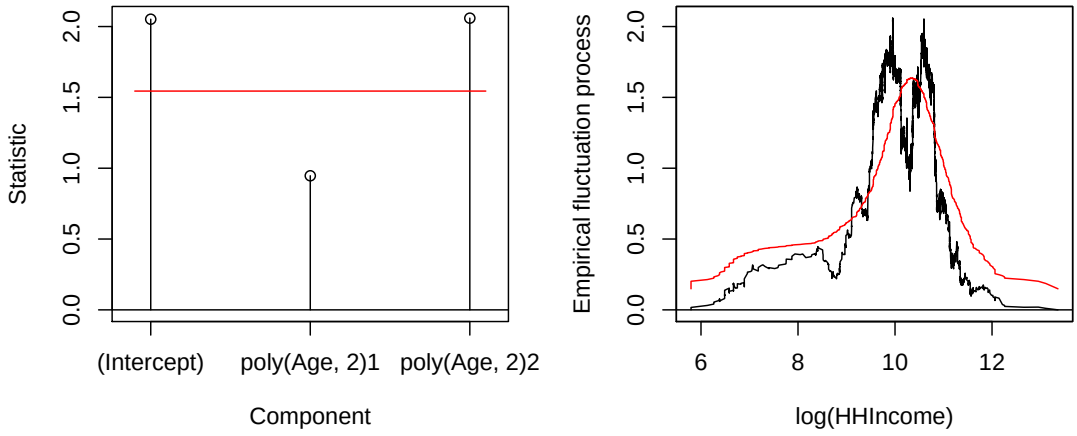


Figure 3: Empirical M-fluctuation process tested with `dmax1` (left) and `dmax2` (right) double maximum functional.

Both functionals `dmax1` and `dmax2` do not provide many new insights compared to `maxBB` and were mainly employed for illustrating how new functionals can easily be created and applied to data by users without writing a new implementation of the full test. In real applications, typically only a single functional will be used (otherwise one would have to correct for multiple testing) and the choice of the functional can and should be guided by knowledge about the problem at hand: it can be used to incorporate prior information about the timing and/or the component of the shift as well as the particular alternative of interest (see Zeileis and Hornik 2003, for a brief discussion).

5.3 Beta regression

To illustrate that the new function `gefp` from can not only be used with the standard `glm` function for generalized linear regression models, we present a brief example for beta regression. This regression method for rates and proportions was suggested by Ferrari and Cribari-Neto (2004) and an implementation in the R language is provided in **betareg** (de Bustamante Simas 2004).

This package can be loaded by

```
R> library(betareg)
```

and it provides the function `betareg` which fits a beta regression model via ML based on a formula specification. Thus, beta regression is based on an estimating function and we just need a function which can extract the ML scores from a fitted `"betareg"` object. As the package does not include such an extractor, we simply provide an `estfun` method in the appendix, based on Equations (8) and (9) in Ferrari and Cribari-Neto (2004). This is already sufficient to compute generalized M-fluctuation processes based on beta regression fitted by `betareg`. Here, we simulate two artificial series which are subsequently assessed by `gefp`. For both series, we simulate 200 observations from a beta distribution with a single shift in the parameters. We follow the parameterization of Ferrari and Cribari-Neto (2004) employing the mean μ and precision ϕ (corresponding to the commonly used shape parameters $\mu\phi$ and $(1-\mu)\phi$). Both simulated series start with the parameters $\mu = 0.3$ and $\phi = 4$ and for the first series μ changes to 0.5 at $t = 0.75$ while ϕ remains constant whereas for the second ϕ changes to 8 at $t = 0.5$ and μ remains constant.

```
R> set.seed(123)
R> y1 <- c(rbeta(150, 0.3 * 4, 0.7 * 4), rbeta(50, 0.5 * 4, 0.5 *
+     4))
R> y2 <- c(rbeta(100, 0.3 * 4, 0.7 * 4), rbeta(100, 0.3 * 8, 0.7 *
+     8))
```

In the following, a generalized M-fluctuation process is fitted to each series and its fluctuation is assessed using the default `maxBB` functional from Equation (6).

```
R> y1.efp <- gefp(y1 ~ 1, fit = betareg)
R> y2.efp <- gefp(y2 ~ 1, fit = betareg)
R> plot(y1.efp, aggregate = FALSE)
R> plot(y2.efp, aggregate = FALSE)
```

The upper panel of Figure 4 depicts the resulting process along with its boundaries for the series `y1`. This gives exactly the desired results: while the process for ϕ exhibits only moderate fluctuation, the process for the intercept (corresponding to μ) crosses its 5% level boundary and thus signals a significant parameter instability. From the triangular shape it can be seen that the mean of the series increased at about $t = 0.75$. Similarly, the results for `y2` reflect exactly how the data was generated. Now, the process for μ fluctuates only moderately whereas the triangular shape of the process for ϕ signals a clear and significant increase in ϕ at about $t = 0.5$.

6 Conclusions

It is discussed how the ideas and flexibility of a conceptual method can be translated into econometric software by exploiting language features such as object orientation, functions as first class objects and nested lexically scoped functions and by incorporating/exporting modular re-usable functions.

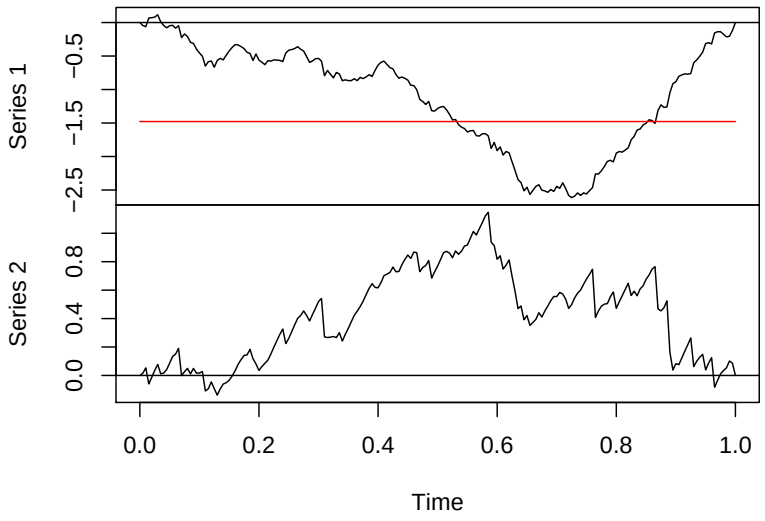
The implementation of the class of generalized M-fluctuation tests for structural change in the R language for statistical computing and graphics is described some detail. This is based on building blocks provided by other functions or packages which could be user-supplied or well-known and established R functions. The objects computed are again re-usable components which can be easily used in other programs than those intended by the author; for example in tests for cointegration where similar limiting distributions may occur.

The analysis with generalized M-fluctuation tests is carried out in three steps: 1. computation of an empirical fluctuation process, 2. specification of a functional used for significance testing, and 3. visualization and performance of the corresponding significance test. These steps are reflected in **strucchange** in the following way:

1. The function `gefp` computes empirical fluctuation processes (an object of class `"gefp"`) based on possibly user-defined models and corresponding scores. Numerically reliable functions are available for linear regression, generalized linear models and robust regression. Any other model with an M-type estimator can easily be plugged in: all that is needed is the model fitting function and a function to extract the scores from the fitted model. Furthermore, a function for estimating the covariance matrix can be supplied to `gefp`.

2. The function `efpFunctional` takes a specification of an aggregation functional and then simulates automatically a table of critical values and chooses a sensible visualization technique. The resulting `"efpFunctional"` object provides functions for computation of the test statistic and corresponding p value and for process visualization. Suitable objects for frequently used functionals are already available in **strucchange**.

3. The generic functions `plot` and `sctest` have methods for `"gefp"` objects with an argument for `"efpFunctional"` objects allowing for performing the corresponding significance test either visually or by computation of a test statistic and p value.



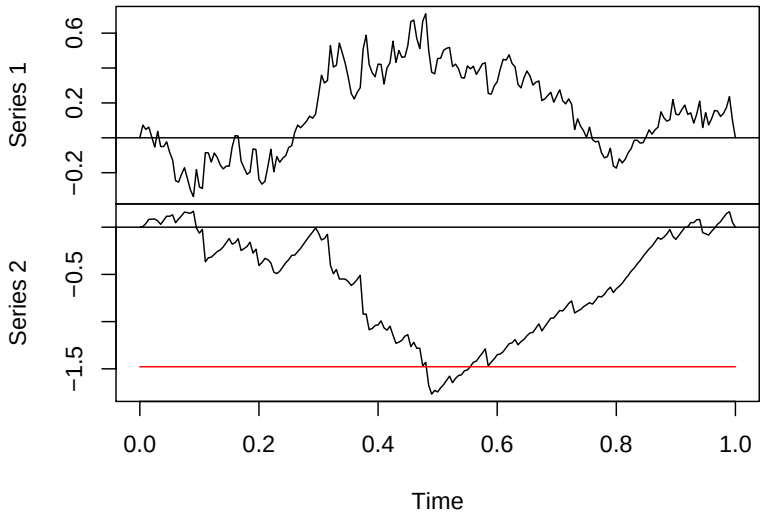


Figure 4: Beta regression, change in μ (upper) and in ϕ (lower).

Furthermore, **strucchange** provides additional functions which can complement the analysis with M-fluctuation tests but lie beyond the scope of this paper: e.g., functions for estimating breakpoints and segmented regression models. More detailed information on all these functions can be found by inspecting the reference manual and, of course, the source code itself which are both contained in the package and freely available from http://CRAN.R-project.org/ where also forthcoming versions of the package will be available.

Computational details

The results in this paper were obtained using R 2.3.0 and the packages **strucchange** 1.2-12, **zoo** 1.0-6, **sandwich** 1.1-1 and **betareg** 1.1

Acknowledgements

The research of Achim Zeileis was supported by the Austrian Science Foundation (FWF) under grant SFB#010 ('Adaptive Information Systems and Modeling in Economics and Management Science').

We are thankful to Kurt Hornik and to the participants of the Computational Management Science Conference and Computational Econometrics Workshop 2004 in Neuchâtel for helpful comments and discussions. Furthermore, we are grateful to editor Erricos Kontoghiorghes, an anonymous associate editor and two anonymous referees for valuable comments on a previous version that lead to a substantial improvement of the paper.

References

Andrews DWK (1991). "Heteroskedasticity and Autocorrelation Consistent Covariance Matrix Estimation." *Econometrica*, **59**, 817–858.

Andrews DWK (1993). "Tests for Parameter Instability and Structural Change With Unknown Change Point." *Econometrica*, **61**, 821–856.

Andrews DWK, Ploberger W (1994). "Optimal Tests When a Nuisance Parameter is Present Only Under the Alternative." *Econometrica*, **62**, 1383–1414.

Brown RL, Durbin J, Evans JM (1975). "Techniques for Testing the Constancy of Regression Relationships over Time." *Journal of the Royal Statistical Society B*, **37**, 149–163.

Chambers JM, Hastie TJ (1992). *Statistical Models in S*. Chapman & Hall, London.

Cribari-Neto F (2004). "Asymptotic Inference Under Heteroskedasticity of Unknown Form." *Computational Statistics & Data Analysis*, **45**, 215–233.

Cribari-Neto F, Zarkos SG (1999). "R: Yet Another Econometric Programming Environment." *Journal of Applied Econometrics*, **14**, 319–329.

de Bustamante Simas A (2004). **betareg***: Beta Regression*. R package version 1.0-1.

Dolnicar S, Leisch F (2000). "Behavioral Market Segmentation Using the Bagged Clustering Approach Based on Binary Guest Survey Data: Exploring and Visualizing Unobserved Heterogeneity." *Tourism Analysis*, **5**(2–4), 163–170.

Doornik JA (2002). "Object-oriented Programming in Econometrics and Statistics Using Ox: A Comparison with C++, Java and C#." In S Nielsen (ed.), "Programming Languages and Systems in Computational Economics and Finance," pp. 115–147. Kluwer Academic Publishers, Dordrecht.

Ferrari SLP, Cribari-Neto F (2004). "Beta Regression for Modelling Rates and Proportions." *Journal of Applied Statistics*, **31**, 799–815.

Gentleman R, Ihaka R (2000). "Lexical Scope and Statistical Computing." *Journal of Computational and Graphical Statistics*, **9**(3), 419–508.

Hansen BE (1992). "Testing for Parameter Instability in Linear Models." *Journal of Policy Modeling*, **14**, 517–533.

Hjort NL, Koning A (2002). "Tests for Constancy of Model Parameters Over Time." *Nonparametric Statistics*, **14**, 113–132.

Kuan CM, Hornik K (1995). "The Generalized Fluctuation Test: A Unifying View." *Econometric Reviews*, **14**, 135–161.

Leisch F, Rossini AJ (2003). "Reproducible Statistical Research." *Chance*, **16**(2), 46–50.

Lumley T, Heagerty P (1999). "Weighted Empirical Adaptive Variance Estimators for Correlated Data Regression." *Journal of the Royal Statistical Society B*, **61**, 459–477.

McCullough BD, Vinod HD (2003). "Verifying the Solution from a Nonlinear Solver: A Case Study." *American Economic Review*, **93**, 873–892.

Nyblom J (1989). "Testing for the Constancy of Parameters Over Time." *Journal of the American Statistical Association*, **84**, 223–230.

Piehl AM, Cooper SJ, Braga AA, Kennedy DM (2003). "Testing for Structural Breaks in the Evaluation of Programs." *Review of Economics and Statistics*, **85**(3), 550–558.

Ploberger W, Krämer W (1992). "The CUSUM Test With OLS Residuals." *Econometrica*, **60**(2), 271–285.

Ploberger W, Krämer W (1996). "A Trend-Resistant Test for Structural Change Based on OLS Residuals." *Journal of Econometrics*, **70**, 175–185.

Racine J, Hyndman R (2002). "Using R to Teach Econometrics." *Journal of Applied Econometrics*, **17**, 175–189.

R Development Core Team (2005). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-00-3, URL http://www.R-project.org/.

White H (1980). "A Heteroskedasticity-Consistent Covariance Matrix and a Direct Test for Heteroskedasticity." *Econometrica*, **48**, 817–838.

Zeileis A (2004a). "Alternative Boundaries for CUSUM Tests." *Statistical Papers*, **45**, 123–131.

Zeileis A (2004b). "Econometric Computing with HC and HAC Covariance Matrix Estimators." *Journal of Statistical Software*, **11**(10), 1–17. URL http://www.jstatsoft.org/v11/i10/.

Zeileis A, Hornik K (2003). "Generalized M-Fluctuation Tests for Parameter Instability." *Report 80*, SFB "Adaptive Information Systems and Modelling in Economics and Management Science". URL http://www.wu-wien.ac.at/am/reports.htm#80.

Zeileis A, Kleiber C, Krämer W, Hornik K (2003). "Testing and Dating of Structural Changes in Practice." *Computational Statistics & Data Analysis*, **44**(1–2), 109–123. doi:10.1016/S0167-9473(03)00030-6.

Zeileis A, Leisch F, Hornik K, Kleiber C (2002). "strucchange: An R Package for Testing for Structural Change in Linear Regression Models." *Journal of Statistical Software*, **7**(2), 1–38. URL http://www.jstatsoft.org/v07/i02/.

A Beta regression estimating functions

To apply the generalized M-fluctuation test methodology to beta regression as implemented in the **betareg** package (de Bustamante Simas 2004) a function is required that extracts the ML scores from a fitted `"betareg"` object. The easiest way to do so is by providing an `estfun` method based on Equations (8) and (9) in Ferrari and Cribari-Neto (2004).[4]

```
estfun.betareg <- function(x, ...)
{
  ## extract response y and regressors X
  xmat <- x$x
  y <- x$y

  ## extract coefficients
  beta <- coef(x)
  phi <- beta[length(beta)]
  beta <- beta[-length(beta)]

  ## compute y*
  ystar = x$funlink$linkfun(y)

  ## compute mu*
  eta <- xmat %*% beta
  mu <- x$linkinv(eta)
  mustar <- digamma(mu * phi) - digamma((1 - mu) * phi)

  ## compute diagonal of matrix T
  Tdiag <- x$funlink$mu.eta(eta)

  ## compute scores of beta
  rval <- phi * (ystar - mustar) * Tdiag * xmat

  ## combine with scores of phi
  rval <- cbind(rval,
    phi = (mu * (ystar - mustar) + log(1-y) - digamma((1-mu)*phi) + digamma(phi)))

  attr(rval, "assign") <- NULL
  return(rval)
}
```

[4]This code will also be provided to the **betareg** authors.