



*Proceedings of the 3rd International Workshop
on Distributed Statistical Computing (DSC 2003)
March 20–22, Vienna, Austria ISSN 1609-395X
Kurt Hornik, Friedrich Leisch & Achim Zeileis (eds.)
<http://www.ci.tuwien.ac.at/Conferences/DSC-2003/>*

Rserve

A Fast Way to Provide R Functionality to Applications

Simon Urbanek

Abstract

Rserve is a TCP/IP server which allows other programs to use facilities of R from various languages without the need to initialize R or link to the R library. Every connection has a separate workspace and working directory. Client-side implementations are available for popular languages such as C/C++ and Java. Rserve supports remote connection, authentication and file transfer. This paper describes the Rserve concept, compares it with other techniques and illustrates its use on several practical examples.

1 Introduction

The R statistical environment provides a powerful suite of tools for statistical analysis for use in many scientific fields. Its application is not limited to statistical research only, but its modular design allows the use of R and its packages in other projects. Often it is feasible to hide the programming aspects of R from the user and integrate the computing capabilities of R into a customized software designed for a specific target group. Possible examples are web-based servlets that allow the user to analyze his/her data using a fixed process, or software for data visualization which uses R facilities to generate statistical models.

R provides two native interfaces for communication with other applications: a simple standard input/output model or R as a shared library. As we will describe later they are unsatisfactory in many situations, either because of speed concerns or when the host application is not written in the C language. In this paper we propose another method of using R facilities from other applications, called Rserve. It is not limited to specific programming languages and even allows separation of client and R environments. The main concerns while developing the system were speed and ease of use. In section 2 we describe the basic design and functionality of

Rserve while more detailed description of the implementation is given in section 3. In section 4 we compare Rserve to other communication methods, including the Omegahat (Temple Lang, 2000) approach and in section 5 we illustrate the use of Rserve on several basic examples. A real application of Rserve is described in section 6. Concluding remarks and ideas for the future are mentioned in section 7.

2 Basic design and features

The main goal of Rserve is to provide an interface which can be used by applications to perform computations in R. Our experience with other communication methods has shown that there are three main points to be considered when designing a new system: separation, flexibility and speed.

It is important to separate the R system from the application itself. One reason is to avoid any dependence on the programming language of the application, since a native direct interface to R (Chambers, 1998) is usable from the C language only (R Development Core Team, 2003). Another aspect comes from the fact that tight integration with R is more error prone, because the application must take internals of R into account. On the other hand application developers want the interface to be very flexible and make use of most R facilities. Finally speed is a crucial element, because the goal is to provide the user with the desired results quickly, without the need of starting an R session from scratch.

A client/server concept allows us to meet all three key requirements. The computation is done by the Rserve core, which is a server answering requests from clients such as applications. The communication between the Rserve and the client is done via network sockets, usually over TCP/IP protocol, but other variations are also possible. This allows the use of a central Rserve from remote computers, the use of several Rserve by the remote client to distribute computation, but also local communication on a single machine.

One Rserve can serve multiple clients simultaneously¹. Every connection to Rserve obtains its own data space and working directory. This means, that objects created by one connection don't affect other connections in any way. Additionally each connection can produce local files, such as images created by R's bitmap graphics device, without interfering with other connections. Every application can open multiple connections to process parallel tasks.

The data transfer between the application and Rserve is performed in a binary form to gain speed and minimize the amount of transferred data. Intermediate objects are stored in Rserve, therefore only objects of interest need to be transferred to the client. For practical examples, see section 5.

Beside communication with the R core, Rserve has also an integrated authentication and file transfer protocol, which makes Rserve suitable for use on separate machines. User authentication is provided to add a level of security for remote use. File transfer allows copying of files needed for the computation or produced by R from the client to the server and vice versa.

Currently Rserve supports two main groups of commands for communication with R: creation of objects in R and evaluation of R code. Most basic objects, such as numbers, strings or vectors can be constructed via direct object creation. The contents of the objects are sent in binary form from the client to the server.

¹An exception to this rule is the Windows operating system. The reasons and alternative solutions are described in the next section.

This provides an efficient way of transporting data necessary for evaluation. All objects are always passed by value to separate client and server data spaces. This way both the client and the server are free to dispose of the data at any time, preventing crashes which are inherent in other communication methods where the systems share the same data physically.

The second main command group is the evaluation of R code. As opposed to object creation such code is sent in clear text to `Rserve` and is treated as if the code was typed on the console in R. The resulting object of the evaluation can be sent back in binary form to the client if requested. Most R types are supported, including scalar numbers, strings, vectors, lists (hence classes, data frames and so on), lexical objects etc. This allows `Rserve` to pass entire models back to the client. The client may decide to not receive any objects, which is useful while setting up intermediate objects in R which are not directly relevant to the client.

`Rserve` provides two basic error handling facilities. The three possible results of an evaluation are successful evaluation, run-time error in the code and parser error. The status is always returned to the client application to allow corresponding action. Since `Rserve` is just a layer between the application and R it is still possible to influence run-time error handling in R itself e.g. with the `error` option or the `try` command.

A typical use of `Rserve` facilities is to load all necessary data into R, perform computations according to the user input, such as construction of models, and send results back to the application for display. All data and objects are persistent until the connection is closed. This allows an application to open a connection early e.g. when the user first specified the dataset, pass all necessary data to the server and respond to user input by ad-hoc computing of the desired models or estimates. Since the results are not in textual form, no tedious parsing of the results is necessary.

The interface to `Rserve` is modular and documented, allowing access to `Rserve` from any application or programming language which supports sockets, including current scripting and programming languages. We have implemented a client for `Rserve` in pure Java, which interfaces to most facilities of `Rserve` and maps all objects available in `Rserve` into native Java objects or classes. The use of the Java client is illustrated in the examples section.

3 Implementation details

In the previous section we have presented the basic goals, design and features of `Rserve`. In this section we aim to describe the technical implementation details. The information provided here is aimed at developers who want to understand the technical details of the implementation or to write a new `Rserve` client. Others are free to skip this section, the use of `Rserve` is illustrated in the following sections.

`Rserve` uses a client/server design to separate R from the client, to allow distribution of tasks over multiple computers or to provide a central computation node. The communication can be performed over any reliable bi-directional stream, the current implementation uses datagram sockets. This allows communication over TCP/IP or unix sockets. The default TCP port is 6311 but can be modified upon `Rserve` startup.

`Rserve` uses its own message-oriented data transfer protocol over the stream, which is described in the `Rserve` online documentation (Urbanek, 2003). It defines the encoding of all basic data types, such as integers, doubles, strings, but also R's

SEXP - simple expressions. All data types in R are internally represented as SEXP. Any more complex objects, such as arrays, vectors, lists or closures are transported as SEXP. Rserve takes care of the encoding and decoding of the expressions and uses its own storage format. Because of the client/server nature of Rserve all objects are passed by value (exception to this rule are symbols which are passed by name, unless explicitly evaluated).

The same format is used for the encoding of data types in both directions. We decided to define our own data transfer encoding in order to be able to mimic R's SEXP more closely. The definition is platform independent as it defines the format of all primitive types as well. The current version of Rserve supports encoding of most SEXP types, namely all primitive types, logicals, vectors, arrays, lists, LANGSXP, CLOSXP and symbols (as names). Decoding is implemented only for integers, doubles, strings and the corresponding arrays. Other types are likely to follow in the future.

The protocol is message-oriented. Rserve waits for a packet containing the command (e.g. `CMD_eval`) and all associated parameters (e.g. `string "R.version.string"`). Rserve performs the specified action (in this case evaluating `"R.version.string"`) and sends a response packet. The response packet indicates whether the command was successful or not and contains also the required data (in our case a string containing the R version). In case of an error the source of the error is indicated (I/O error, parser or run-time evaluation in R).

The commands supported by the current Rserve can be divided into the following categories:

1. user authentication
2. evaluation of R expressions
3. assignment of values to R symbols
4. file transfer
5. administration (server shutdown)

User authentication allows restricting the use of Rserve to certain users. Rserve currently supports two authentication methods: plain text or unix crypt password. The Rserve user information is stored in a separate file on the server, which is not connected with the system user database. The goal is to prevent unauthorized access to Rservices which operate in remote mode. The transport itself is not encrypted, it is possible to use other tools, such as `ssh` to enhance security. For more detailed discussion concerning R, Rserve and security, please consult the online documentation (Urbanek, 2003).

Evaluation of R expressions is the key functionality of Rserve. It enables the client to execute code in R and to retrieve the result. The result is transported to the client as an encoded SEXP. Unless additional packages are used, any printed output is ignored, only the resulting value is returned. The error handling behavior is dependent on settings in R, such as the `error` option. From the application's point of view the result is the same as if the error occurred in a regular R session using the terminal - the default being to unwind the stack and to return the control to the application along with the corresponding error code.

Assigning of values to R symbols provides a fast way of transporting data to Rserve. Without this feature the only way of passing values was to send the corresponding command string, such as: `"a <- c(1.5,2.4,6.7,3.5,1.2)"`.

This approach is rather clumsy since the application usually stores the values in a binary form and would have to construct the string. Further problems are also caused by special characters and limited string representation of a number. Therefore `Rserve` provides a way of transporting values in encoded form to `Rserve` and assigning them to R symbols.

`Rserve` could be extended by adding decoding support for more `SEXP` types. This would allow invocation of R commands in the style of `.Call` function. This functionality should be available in the next versions of `Rserve`.

Since `Rserve` can be used in remote mode, where the server and the client run on different machines, it may be convenient to transfer files, such as data sets or images generated by R from the server to the client or vice versa. `Rserve` provides a simple file transport facility for this purpose. The supported facilities comprise opening, creating, reading, writing and deleting files.

Finally the last group provides the shutdown command for server administration. Although the server responds to the usual shutdown signals, such as `TERM` and `KILL`, the shutdown command provides the most graceful termination. New connections are not accepted, but all current connections are kept open until closed by the client, then the server terminates.

So far we have illustrated what happens while a client is connected to the `Rserve`. One of our main goals was to eliminate R initialization delay and to provide a separate data space and working directory for each connection. Therefore we need to describe what happens when a new connection is accepted.

`Rserve` is linked to the R shared library and it initializes R during its startup. Now we have an initialized R waiting for commands. `Rserve` uses `fork` to create a new, initialized process as soon as a new incoming connection arrives. For most current operating systems `fork` has little overhead, since the same memory is used for the code and data segment is copied on write only. This allows very fast spawning of R processes for the clients. At the same time this method guarantees that each connection receives a clean, separate data space, unpolluted by previous connections which is independent of all other R instances.

Before answering queries from the client `Rserve` creates a new working directory for the connection. The root of the working directories is configurable (default is `/tmp/Rserv`) and each subdirectory is of the form `connX` where `X` is a decimal number unique to the connection. Empty working directories are removed once the connection has been closed. The reason for retaining non-empty directories is that a local application (e.g. web server) may want to access the generated data (e.g. bitmap images previously generated) even after the connection was closed and is then responsible for their removal. This feature may become configurable at some later point if necessary.

`Rserve` initializes an incoming connection by sending an identification string of 32 bytes which describes the basic capabilities of the server. Therefore listening `Rserve`s are easily identifiable as they send 32 bytes of which the first four are "Rsrv".

`Rserve` was primarily designed for unix operating systems, because those are mainly used for network servers. `Rserve` can be also used on computers running the Windows operating system, but several restrictions apply. The main difference is the inability to use the `fork` command to spawn new instances of `Rserve` quickly. Windows provides no such facilities. Although `Rserve` supports threads, R does not and therefore there is no alternative but to use separate `Rserve` instances for separate connections. Therefore the Windows version of `Rserve` supports only one connection at a time. Any subsequent connections to the same `Rserve` share the same working

directory and data space. In this case Rserve doesn't change the working directory.

Depending on the client there are several approaches for using Rserve in a Windows environment. Applications which use one Rserve instance at a time are free to launch their own instance, use it and shut it down upon completion. This is sufficient for most applications. More sophisticated applications can initialize a pool of parallel Rserve instances and distribute computation among them, launching new instances when necessary. In Windows this can be easily done by an external application, therefore we decided not to incorporate this functionality directly in Rserve.

4 Comparison with other methods

Rserve aims to complement the variety of available methods for communication to R, not to replace them. Native API for communication with R is defined on the level of the C or FORTRAN languages, excluding other languages unless some kind of a bridge is used, specific for each language. Rserve provides an interface which is defined independently of any programming or scripting language.

A console-based interface, where commands for R are stored in a file and written into another file (also known as batch mode) is currently used by several applications, such as VASMM (Masaya Iizuka and Tanaka, 2002). It is very slow, because a new instance of R has to be started for each request. Results are usually stored in textual form, which is not suitable for interprocess communication and requires a parser at the application's end. This may not be a problem for some scripting languages such as perl, but it is a problem for other languages such as Java.

Rserve still provides a way of capturing textual output if necessary, although the preferred method is binary transfer. In turn Rserve has very little overhead for each new connection, because it doesn't need to initialize a new instance of R.

Since our main applications of Rserve involve the Java client for Rserve, we also compared Rserve to the SJava interface from the Omegahat project (Temple Lang, 2000). SJava is conceptually far more flexible than Rserve, because it allows calling both R from Java and Java from R. Rserve implies that Java is the controlling application and unlike SJava it has no concept of callbacks. As Rserve provides computational facilities for the client and every action is initiated on the Java side, callbacks are in fact undesirable, since R is not thread-safe.

Let us compare Rserve with the R-from-Java part of SJava, because they are based on very similar philosophies. Rserve can be used remotely, because objects are copied when necessary. This approach allows distributed computing on multiple machines and CPUs. SJava works only locally since it embeds R into Java via JNI. Due to the fact that there is no synchronization between Java and R, and given that R is not multi-thread safe, it is fatal to make more than one call from Java into R. The application developer is responsible for proper synchronization when using SJava. Rserve performs this synchronization by design and also allows the use of multiple concurrent connections. SJava allows passing of object references, which can lead to serious problems and crashes if utmost care is not taken.

Both SJava and Rserve support conversion of basic object types between Java and R. Rserve provides much a wider variety of objects passed to Java by encapsulating all native *SEXP* (simple expressions) of R into a Java class. In SJava conversion of complex types is supported, but the developer must implement his own class converters.

Probably one of the main disadvantages of `SJava` is that it does not run out-of-the-box. The code is dependent on the hardware and operating system used, as well as the `Java` implementation. In general it is very hard to setup and the solution cannot be deployed with the application. `Rserve` comes as a regular `R` source package for unix platforms and as a binary executable for Windows. The client side of `Rserve` needs no special setting and is platform independent, since it is written in pure `Java`, currently requiring only the JDK 1.1 specification. The `Rserve` client classes can be easily deployed with any `Java` program and no third party software is necessary.

Finally `R` has its own set of functions for socket communication, therefore it should be possible to build a pure `R` program mimicking the same functionality as `Rserve`. Although this is true, such an `R` program would only be able to serve one connection at a time and would lack separate workspaces. The use of the serializable format of `R` instead of the `Rserve` protocol was also suggested, but the serialization is known only to `R` and therefore the application would have to implement the full serialization protocol. Only limited documentation was at our disposal when the decision was made, therefore we decided to use our own binary protocol.

5 Using Rserve

`Rserve` itself is provided as a regular `R` package and can be installed as such. The actual use is not performed by the `library` command, but by starting the `Rserve` executable (Windows) or typing `R CMD Rserve` on the command line (all others). By default `Rserve` runs in local mode with no enforced authentication. Once the `Rserve` is running any applications can use its services.

All of our applications using `Rserve` represent `Java` programs which use `R` for computation, therefore we will show examples using the `Java` client for `Rserve`. The principles are identical when using other `Rserve` clients, therefore using `Java` as the starting point poses no limitation.

Before plunging into real examples, let us consider the minimal “hello world” example:

```
Rconnection c = new Rconnection();
REXP x = c.eval("R.version.string");
System.out.println(r.asString());
```

The code has the same effect as typing `R.version.string` in `R`. In the first line a connection to the local `Rserve` is established. Then the `R` command is evaluated and the result stored in a special object of the class `REXP`. This class encapsulates any objects received or sent to `Rserve`. If the type of the returned objects is known in advance, accessor methods can be called to obtain the `Java` object corresponding to the `R` value, in our case a regular `String`. Finally this string is printed on the standard output.

The following code fragment illustrates the use of slightly more complex native `Java` types:

```
double[] d = (double[]) c.eval("rnorm(100)").getContent();
```

This single line in `Java` provides an array of 100 doubles representing random numbers from the standard normal distribution. The numeric vector in `R` is automatically converted into `double[]` `Java` type. In cases where no native `Java` type exists,

Rserve Java client defines its own classes such as `RList` or `RBool`². This approach makes the use of Rserve very easy.

As a first more practical example we want to calculate a Lowess smoother through a given set of points. The Java application lets the user specify the data allowing interactive changes of the points, displays a regular scatter plot and needs coordinates of the smoother to be obtained from R.

One way of obtaining such a result would be to construct a long string command of the form `lowes(c(0.2,0.4,...), c(2.5,4.8,...))` and using the `eval` method to obtain the result. This is somewhat clumsy, because the points usually already exist in a `double` array in the Java application and the command string must be constructed from these. An alternative involves constructing objects in R directly. The following code shows the full Lowess example:

```
double[] dataX,dataY;
...
Rconnection c = new Rconnection();
c.assign("x",dataX);
c.assign("y",dataY);
RList l = c.eval("lowess(x,y)").asList();
double[] lx = (double[]) l.at("x").getContent();
double[] ly = (double[]) l.at("y").getContent();
```

First the Java application defines the arrays for the data points *dataX* and *dataY*. The application is responsible for filling these arrays with the desired content. Then we assign the contents of these arrays to R variables *x* and *y*. The `assign` command transfers the contents in binary form to Rserve and assigns this content to the specified symbol. This is far more efficient than constructing a string representation of the content.

Once the variables are set in R we are ready to use the `lowess` function. It returns a list consisting of two vectors *x* and *y* which contain the smoother points. The `RList` object provides the method `at` for extraction of named entries of a list. Since lists may contain entries of different types, the object returned by the `at` method is of the class `REXP` whose content can be cast into `double[]` in our case. The result can now be used by the Java application.

More complex computations can be performed even without transmission of resulting objects. This is useful when defining functions or constructing complex models. Model objects are usually large, because they contain original data points, residuals and other meta data. Although they can be transferred to the client, it is more efficient to retain such objects in R and extract relevant information only. This can be done by using the `voidEval` method which does not transfer the result of the evaluation back to the client:

```
c.assign(y, ...) ...
c.voidEval("m<-lm(y~a+b+c)");
double [] coeff =
    (double[]) c.eval("coefficients(m)").getContent();
```

In the above example a linear model is fitted, but its content is not passed back to the client. It is stored in an object in R for later use. Finally the coefficients are extracted from the model and passed back to the Java application.

²Java's boolean type has no support for NA missing values, therefore it cannot be used to directly represent the logical type in R.

So far we used `Rserve` in local mode only. Extension to remote `Rserve` connections is possible without code changes, except for additional parameters to the `Rconnection` constructor, specifying the remote computer running the `Rserve`. For details about the use of remote authentication, error handling and file transfer, consult the documentation supplied with the `Rserve` and the `Java` client. The use is again straight-forward, since native `Java` facilities, such as input/output streams are used.

6 Example

In the following we want to describe a real-life application of `Rserve`. The example features `Klimt` (Urbanek and Unwin, 2001), a software for visualization and analysis of trees and forests. `Klimt` is written entirely in `Java` and provides numerous interactive facilities for visualization of tree models and analysis of associated data. `Klimt` can be used as a stand-alone application, but it requires `R` for the construction of tree models. Therefore it needs a way of communicating with `R` to perform the necessary computations.

There are four tasks for which `Klimt` connects to a `Rserve`: initialization, construction of a tree from the open data set, construction of tree branches when the user interactively modifies a tree and finally construction of derived variables.

When initializing `R` by opening the `Rserve` connection, `Klimt` checks the version of `R` and loads the necessary libraries for tree construction - `tree` or `rpart` depending on the user's choice. Before the first tree is generated or the first variable is used, `Klimt` stores the entire data set in `R` by assigning each variable of the data set into `R` objects of the same name. For tree construction `Klimt` simply evaluates an `R` expression of the form: `"tree("+formula+", "+parameters+")$frame"`. The resulting object contains a data frame which entirely describes the tree. This information is converted by `Klimt` into an internal representation of a tree. The formula is generated from the items selected by the user from a list. Optional parameters can be specified by the user.

It is recommended to wrap the evaluated expression in the `try` function. The resulting object is then either the requested tree, or a string containing the error message if the command was not successful. In `Klimt` the actual code looks like this:

```

REXP r=c.eval("try(tree("+formula+", "+parameters+")$frame)");
if (r.getType()==REXP.XT_STRING) {
    String error=r.asString();
    ...
} else {
    SNode root=convertTree(r.asList());
    ...
}

```

Here `SNode` is the internal recursive representation of a tree in `Klimt`. A similar approach is used for interactive tree splitting. The user interactively specifies the split, resulting in two nodes. Two subsets corresponding to the interactively created nodes are used, one tree is grown for each node and attached to its parent node. The main advantage is that the connection is held open and therefore the data set doesn't need to be re-transmitted to `Rserve`.

Finally derived variables can be created by evaluating an expression supplied by the user and stored in the requested variable:

```
REXP r=c.eval("try("+varName+" <- "+expr+"");
```

If the expression supplied by the user is correct, then the result must be an array of the same length as the data set in Klimt. Since the variables are stored directly in R, expressions of the form $v1/v2+v3$ deliver the expected result if the data set contains the variables $v1$, $v2$ and $v3$.

7 Conclusions

Rserve complements the family of interfaces between applications and R by providing a fast, language-independent and remote-capable way of using all facilities of R from other programs. Due to a clean separation between R (server) and the application (client), internal data manipulation on one side cannot affect the other. Using network sockets for the communication ensures platform and software independence of the client and the server. At the same time restriction to local use is also possible, requiring no physical network.

For concurrent connections Rserve offers both data and file space separation between connections. Each new connection is accepted almost immediately without the need for initializing R engine. Integrated file transfer protocol allows the use of remotely created files, such as plot bitmaps created by R. User authentication is provided for a level of security, especially when used in remote mode. This concept is suitable for distributed computing and load balancing.

The supplied Java client provides an easy embedding of R facilities into Java programs. Evaluation and transfer of most types from R to the application is provided, including complex objects such as models. All basic types are automatically converted to corresponding Java classes.

Rserve is very versatile, since it poses no limit on the facilities of R used. Although Rserve allows the execution of all R commands, the user should avoid any commands involving the GUI or console input, since Rserve has no console and there is no guarantee that it has any GUI at all. An exception are applications that provide their own copy of Rserve and have control over the way Rserve is started. Typical uses of Rserve include interactive applications using R for model construction (see KLIMT project) or web-servlets performing online computations.

As of now only basic types, such as numbers, strings and vectors hereof, can be assigned directly to R objects. The framework allows the transfer of arbitrarily complex types supported by the REXP, but the Rserve side is not fully implemented yet. Only transfer of evaluated objects supports all common expression types.

Rserve currently provides two client implementations: for Java and C languages. The Rserve protocol is well defined and allows the implementation of further clients in other programming or scripting languages when needed.

Rserve was tested on Linux, Mac OS X and Windows operating systems. The Windows version is the only restricted one, because there is no possible way of spawning new instances of R quickly in Windows. If all a Windows application needs is non-concurrent Rserve connections, it can provide its own copy of the Rserve binary, which will automatically find the last installed R and use it for computations, preventing clashes with other applications.

The Rserve project is released under GPL software license, which means that it can be modified or enhanced if necessary. The current Rserve is already used by several projects and is being enhanced as needs arise. For details and recent development, please visit the Rserve project page:

<http://stats.math.uni-augsburg.de/Rserve>

References

John M. Chambers. *Programming with Data. A Guide to the S Language*. Springer-Verlag, NY, 1998.

Tomoyuki Tarumi Masaya Iizuka, Yuichi Mori and Yutaka Tanaka. Statistical software VASMM for variable selection in multivariate methods. In *COMPSTAT 2002 Proceedings in Computational Statistics*, pages 563–568. Physica, Heidelberg, 2002.

Duncan Temple Lang. The Omegahat environment: New possibilities for statistical computing. *JCGS*, 9(3), 2000.

R Development Core Team. Writing R extensions, 2003. URL <http://cran.r-project.org/doc/manuals/R-exts.pdf>.

Simon Urbanek. Rserve online documentation, 2003. URL <http://stats.math.uni-augsburg.de/Rserve>.

Simon Urbanek and Antony R. Unwin. Making trees interactive - KLIMT. In *Proc. of the 33th Symposium of the Interface of Computing Science and Statistics*, 2001.

Affiliation

Simon Urbanek
Department of computer oriented statistics and data analysis
University of Augsburg
Universitätsstr. 14
86135 Augsburg
Germany
E-mail: simon.urbanek@math.uni-augsburg.de