Index

What is this?
Running the examples
The installation

JRI

Installing JRI
Example 1: The minimal program
Example 2: Some feedback
Example 3: R with a GUI!

JavaGD

Installing JavaGD
Example 1: The JavaGD device
Example 2: Your own JavaGD device
Example 3: Plotting to your own GDCanvas

Good bye, and thanks for all the fish

What is this?

When I came across JRI and JavaGD in my search for an interface from Java to R, the two first things that crossed my mind were:

- Wow, what a cool thing!
- Now, where the hell is the documentation?

Now that I have done a few things with JRI and JavaGD, both feelings have deepened: The fascination for all the cool stuff that can be done and the desperate longing for an example-filled and simple cookbook for this interface. Well, so here goes.

Just to make sure I am not misunderstood: This is not the official documentation for the JRI or JavaGD. I will be showing what I learned in the hope of making life easier for others, but I know that many of the things I'm showing here can be done better. As soon as I find how, I'll update the page. Also, I know that this is by no means complete. I'm in the process of learning, and I will (hopefully) be adding to this page in the process.

Also, make sure to have a look at the JavaDoc for the JRI.

So, if you want to ask a question, poke me in the eye for my stupid code, or suggest an improvement or solution to an interesting problem, go ahead and mail me at falka777 (at) gmx.de.

Running the examples

There are two ways of running the examples, both described at the end of Example 1: The minimal program.

Make sure that you read these instructions. The easier of the two ways is probobaly using the run script provided with JRI.

The installation

One thing before I get down to the interesting stuff: I'm using Linux. I'm not interested in Windows, and unfortunately, I don't have a mac. So, unless I'll be forced to make JRI and JavaGD work with Windows someday, everything here will be aimed at Linux. This is probobaly only interesting for the installation though, so no harm done, eh?

JavaGD

Installing JRI

Needed:

- A package that provides libR.so (r_core_base for Debian)
- JDK 1.5

Paths:

- **LD_LIBRARY_PATH**: [path to libjvm.so (...jdk1.5.0_06/jre/lib/i386/client/)]:[path to the jre libs (...jdk1.5.0_06/jre/lib/i386/)]
- **JAVA_HOME**: [Path to your Java home (...jdk1.5.0_06)]
- **R_HOME**: [(/usr/lib/R on Debian, should contain the subdirs: bin, etc, lib, library, modules, share, site-library)]
- **R_INCLUDE_DIR**: [Path to your R includes: RConfig.h etc. (/usr/share/R/include on Debian)]
- **R_SHARE_DIR**: [Path to the R share dir: $R_INCLUDE_DIR/../share (/usr/share/R/share on Debian)]
- **R_DOC_DIR**: [Path to the R doc dir: $R_INCLUDE_DIR/../doc (/usr/share/R/doc on Debian)]

In case you didn't know: You should set the paths like this: "export R_HOME=pathtoset" or, in case of LD_LIBRARY_PATH: "export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:path1:path2". At least if you're using bash.

After you have downloaded the most recent version of JRI from the developer's homepage, tar -xvzf it to some directory, let's call it Djri.

Now, go to Djri and run ./configure followed by make. In case configure can't find something it needs, install it.

To test the installation, run "./run rtest" and "./run rtest2" from Djri. Everything should work just fine now.

Now honestly, nothing ever works "just fine" the first time you try it - not on Linux anyway. So, what can have gone wrong?

The first common problem is Java telling you that it can't load a shared object. In this case, simply make sure that the given .so is on the **LD_LIBRARY_PATH** and try again.

If problems prevail, make sure that really all shared objects needed by libjri.so are on the **LD_LIBRARY_PATH**: in Djri, run "ldd libjri.so". The listed shared objects should not be pointing to "not found", obviously.

Other problems may be caused by running the wrong version of Java. On many Linux systems, kaffeine will be installed as the default JVM. To check this, run "java -version". It should say something like 'java version "1.5.0_06"'. If it produces a lot of helpful information instead, scroll up and check the first lines of output. Pretty likely it will say something about kaffeine. In this case, set your PATH to point to your sun java executable before it points to anything else (in my case: export PATH=/home/wojtek/Dtools/Djava /jdk1.5.0_06/jre/bin/java:$PATH).

Example 1: The minimal program

The first thing that I want to draw your attention to (again) is that there is a JavaDoc for the JRI.

Now, let's do some fun stuff! Here's some code for your first minimal JRI example. Copy this into a file named Example1.java

```java
import org.rosuda.JRI.REXP;
import org.rosuda.JRI.Rengine;

public class Example1 {
  public static void main(String[] args) {
    System.out.println("Creating Rengine (with arguments)");
    //If not started with --vanilla, funny things may happen in this R shell.
    String[] Rargs = {"--vanilla"};
    Rengine re = new Rengine(Rargs, false, null);
    System.out.println("Rengine created, waiting for R");
    // the engine creates R is a new thread, so we should wait until it's
    // ready
    if (!re.waitForR()) {
      System.out.println("Cannot load R");
      return;
    }

    //In R, call rnorm(4), which generates 4 numbers from a standard
    //normal distribution.
    //The result will be stored in re.
    REXP rn = re.eval("rnorm(4)");

    //The data type REXP provides functions for converting to different
    //data types. In this case we know that rnorm(4) must have returned
    //an array of doubles, so we know what to convert to:
    double[] rnd = rn.asDoubleArray();

    //Let's see the variables.
    for(int i=0; i<rnd.length; i++)
      System.out.print(rnd[i] + " ");

    System.out.println();

    //And that's it! Easy, huh?
    re.end();
    System.out.println("Bye.");
  }
}
```

Now, compile it with the following command:

javac -classpath [path to JRI.jar (Djri/src)]:. Example1.java

To run the example, type:

java -Djava.library.path=[path to libjri.so (Djri)]:[path to libR.so (/usr/lib/R/lib/ on Debian)] -classpath (path to JRI.jar (Djri/src)):. Example1

Of course, if you're using Eclipse, it's all a lot easier - right click on Example.java, go to "Run as"->"Run...", and enter the appropriate values in the Arguments, Classpath and Environment tabs.

An alternative way of running the example is going to your JRI installation directory and using the run script (remember testing JRI with ./run rtest?). There, set **CLASSPATH** to the path in which your compiled example resides, and do "./run Example1"

You should now see output like this:

```
Creating Rengine (with arguments)
Rengine created, waiting for R
-0.7836722026166387 0.046467322350697844 -1.0499623364899335 1.4009138834702612
Bye.
```

Example 2: Some feedback

The first thing that I want to draw your attention to (again) is that there is a JavaDoc for the JRI (I can't stress this enough).

Now that we have R running, it might be nice to see what it has to tell us. For example, if we load a library, it might be nice to know if this actually succeeded, and if it didn't, we might want to know why. All this goes to the R console. And fortunately, there is a simple way of accessing the R console from Java! We just have to give the Rengine something like an EventListener. In our case, it is a class that implements RMainLoopCallbacks. The example below should be pretty much self explaining:

```java
import org.rosuda.JRI.RMainLoopCallbacks;
import org.rosuda.JRI.Rengine;

public class Example2 {
  public static void main(String[] args) {
    String[] Rargs = { "--vanilla" };
    //This time, give the R engine a callback listener.
    Rengine re = new Rengine(Rargs, false, new CallbackListener());
    if (!re.waitForR()) {
      System.out.println("Cannot load R");
      return;
    }

    // Say cheese!
    re.eval("cat('Hello from R!\n')");

    re.end();
  }

  private static class CallbackListener implements RMainLoopCallbacks {
    public void rWriteConsole(Rengine arg0, String arg1) {
      System.out.print(arg1);
    }

    public void rBusy(Rengine arg0, int arg1) {  }

    public String rReadConsole(Rengine arg0, String arg1, int arg2) { return null; }

    public void rShowMessage(Rengine arg0, String arg1) { }

    public String rChooseFile(Rengine arg0, int arg1) { return null; }

    public void rFlushConsole(Rengine arg0) { }

    public void rSaveHistory(Rengine arg0, String arg1) { }

    public void rLoadHistory(Rengine arg0, String arg1) { }
  }
}
```

Running this (in the same way as described in the first example) should give you this output:

```
R : Copyright 2006, The R Foundation for Statistical Computing
Version 2.3.0 (2006-04-24)
ISBN 3-900051-07-0

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

Hello from R!
```

Example 3: R with a GUI!

Now, let's do something more exciting. Let's make a small Java GUI simulating the following situation: You have an R script that takes a while to make some computations. To let the user know what it's doing, it prints a small progress bar: one # for every one of the 20 operations it does.

So, what do we need? At least a start button and a progress bar to show us the progress that R prints to the console in the GUI. To achieve this, we need a JFrame to host the button and the progress bar. It's practical to make the JFrame the handler for the R callbacks too - actually, it's best to have the JFrame handling all the R stuff in this case.

The code hasn't changed much from the last example - the practically unchanged creation of the Rengine can be found in initGUI(). The remaining parts are GUI handling and should be pretty self explanatory.

The only real gotcha is the place when rWriteConsole adds 1 to the value of the progress bar. To get the progress bar to update properly, a call to paintImmediately() is necessary - just a repaint() won't do.

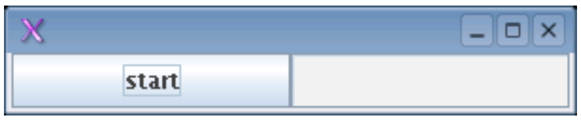
```java
import java.awt.GridLayout;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowEvent;

import javax.swing.JFrame;
import javax.swing.JButton;
import javax.swing.JProgressBar;

import org.rosuda.JRI.RMainLoopCallbacks;
import org.rosuda.JRI.Rengine;

public class Example3 {
  /**
   * Create the PbarFrame and make it visible. The PBarFrame takes care of
   * everything from now on.
   */
  public static void main(String[] args) {
    PbarFrame pf = new PbarFrame();
    pf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pf.setVisible(true);
  }

  /**
   * This JFrame displays the GUI (a button and a progress bar). It also takes
   * care of creating the Rengine and it takes the callbacks.
   */
  private static class PbarFrame extends JFrame implements ActionListener, RMainLoopCallba
    private static final long serialVersionUID = 1L;

    private JProgressBar _jpb;

    private JButton _start;

    private Rengine _re = null;

    // This is set to true when the progressbar is supposed to progress.
    private boolean _progressing = false;

    /**
     * Constructor.
     */
    public PbarFrame() {
      super();
      initGui();
      initR();
    }

    /**
     * Initialise the GUI - create the start button and the progress bar, put
     * them into the content pane.
     */
    private void initGui() {
      getContentPane().setLayout(new GridLayout(1, 2));

      // Create a button and add it to the GUI
      _start = new JButton("start");
      _start.addActionListener(this);
      getContentPane().add(_start);
      // Create a new Prograssbar with minimum value 0 and maximum value 20
      _jpb = new JProgressBar(0, 20);
      _jpb.setValue(0);
      getContentPane().add(_jpb);

      pack();
    }

    /**
     * Initialise the R system - create an R engine and source the file that contains the
     * implementation of the R function that draws us a nice progress bar.
     */
    private void initR() {
      System.out.println("Creating Rengine (with arguments)");
      String[] Rargs = { "--vanilla" };
      _re = new Rengine(Rargs, false, this);
      System.out.println("Rengine created, waiting for R");
      if (!_re.waitForR()) {
        System.out.println("Cannot load R");
        return;
      }

      // Load the file with the definition of the function my.pbar()
      _re.eval("source('pbar.R')");
    }

    /**
     * When the start button gets clicked, call the R function that will draw the progress
     * Before that happens, set the value of the GUI progressbar to 0 and set _progressing
     * true, because the progressbar only gets updated if _progressing is true.
     */
    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == _start) {
        _jpb.setValue(0);
        _progressing = true;
        // Tell R to draw a progress bar.
        _re.eval("my.pbar()");
      }
    }

    /**
     * If R writes something to the console, print it. If _progressing is true, also progr
     * progress abr by 1, and repaint it (paintImmediately has to be used, if you use repa
     * will experience a frozen progress bar that will update all steps at once when R ret
     * the function call)
     */
    public void rWriteConsole(Rengine arg0, String arg1) {
      System.out.print(arg1);
      if (_progressing) {
        _jpb.setValue(_jpb.getValue() + 1);
        _jpb.paintImmediately(0, 0, _jpb.getSize().width, _jpb.getSize().height);
      }
    }

    /**
     * When the window gets closed, we want to cleanly exit R before closing the applicati
     */
    protected void processWindowEvent(WindowEvent e) {
      _re.end();
      super.processWindowEvent(e);
    }
```

```
    public void rBusy(Rengine arg0, int arg1) {
    }

    public String rReadConsole(Rengine arg0, String arg1, int arg2) {
      return null;
    }

    public void rShowMessage(Rengine arg0, String arg1) {
    }

    public String rChooseFile(Rengine arg0, int arg1) {
      return null;
    }

    public void rFlushConsole(Rengine arg0) {
    }

    public void rSaveHistory(Rengine arg0, String arg1) {
    }

    public void rLoadHistory(Rengine arg0, String arg1) {
    }
  }
}
```

Apart from this code, you'll need to copy the following lines into a file called "pbar.R" that has to reside in the same directory as Example3.java:

```
my.pbar <- function() {
 for(i in 1:20) {
  for(j in 1:100000) {
  }
  cat('#')
 }
}
```

Once you have copied this code to Example3.java and compiled and started it just like the first example, you should see something like this:



When you click the start button, the progress bar should slowly advance. If it's too slow or too fast, modify the range for j in pbar.R.

If you are using the run script described in example 1 to run this example, make sure that pbar.R is in the same directory as the run script, or adjust the path in Example3.java accordingly.

JavaGD

In case you want to run these examples with the run script (as described in the first example for JRI), make sure that javaGD.jar is in your classpath (not the path to javaGD.jar, but the archive, e.g. "export

CLASSPATH=$CLASSPATH:/usr/local/lib/R/site-library/JavaGD/java/javaGD.jar"). Other than that, the run script works just as described before.

Installing JavaGD

Installing JavaGD should just be a matter of typing install.packages("JavaGD") at your R prompt and waiting - have a look at the JavaGD homepage. If errors occur, the same holds as in "Installing JRI". You have to carefully check that all that is needed is on the LD_LIBRARY_PATH, and that you have Java installed and all necessary paths set correctly.

Example 1: The JavaGD device

Example is a big name for this. Enter these commands at your R console and you will plot to a JavaGD device. If this doesn't work, something has gone wrong with your installation, though I can hardly imagine what could go wrong at this stage:

```
library(JavaGD)
JavaGD()
plot(c(1,5,3,8,5), type='l', col=2)
```

Example 2: Your own JavaGD device

This is what seems to be causing some confusion - trying to get R to draw to your own implementation of the JavaGD device, not the default one. From what I understand, the function JavaGD() calls the implementation of JavaGD found in the class defined by the environment variable **JAVAGD_CLASS_NAME**. So if you want R to draw to your own implementation of JavaGD, just create a class that extends GDInterface, set **JAVAGD_CLASS_NAME** to the name of your class (notation: my/package/JavaGDClass) - the easiest way of doing this is having the Rengine evaluate "Sys.putenv('JAVAGD_CLASS_NAME'='my/package/JavaGDClass')" - and you're good to go. This is demonstrated in the following example:

File: JavaGDExample1.java

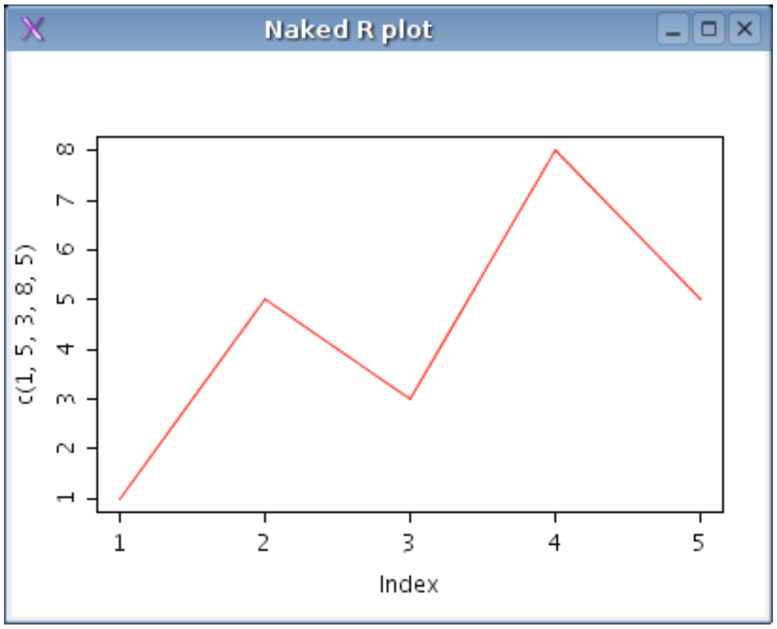
```java
import org.rosuda.JRI.Rengine;

public class JavaGDExample1 {

  public static void main(String[] args) {
    Rengine re;
    String[] dummyArgs = new String[1];
    dummyArgs[0] = "--vanilla";
    re = new Rengine(dummyArgs, false, null);
    re.eval("library(JavaGD)");

    // This is the critical line: Here, we tell R that the JavaGD() device that
    // it is supposed to draw to is implemented in the class MyJavaGD. If it were
    // in a package (say, my.package), this should be set to
    // my/package/MyJavaGD1.
    re.eval("Sys.putenv('JAVAGD_CLASS_NAME'='MyJavaGD1')");
    re.eval("JavaGD()");
    re.eval("plot(c(1,5,3,8,5), type='l', col=2)");

    re.end();
  }
}
```

File: MyJavaGD1.java

```
import javax.swing.JFrame;

import org.rosuda.javaGD.GDCanvas;
import org.rosuda.javaGD.GDInterface;

/**
 * This is a minimal reimplementation of the GDInterface. When the device is opened,
 * it just creates a new JFrame, adds a new GDCanvas to it (R will plot to this GDCanvas)
 * and tells the program to exit when it is closed.
 */
public class MyJavaGD1 extends GDInterface {
  public JFrame f;

  public void gdOpen(double w, double h) {
    f = new JFrame("JavaGD");
    c = new GDCanvas(w, h);
    f.add((GDCanvas) c);
    f.pack();
    f.setVisible(true);
    f.setTitle("Naked R plot");
    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
  }

}
```

Compile and run as you already did in example 1 for JRI. Just make sure that this time, you also have javaGD.jar on your classpath. When you install the JavaGD package in R, javaGD.jar is put into your lib.loc/JavaGD/java/ (/usr/local/lib/R/site-library/JavaGD/java/ in Debian)

You should now see something like this:



Notice that the title says "Naked R plot" - just as we set it in MyJavaGD1.java, not "JavaGD (1) *active*", as would be the title in the default JavaGD implementation. Congrats, you just made R plot to your own JavaGD device!

Example 3: Plotting to your own GDCanvas

Basically, this is pretty self explaining. Here's an example of how you can neatly integrate the plot from R into your Java GUI:

File: JavaGDExample2.java

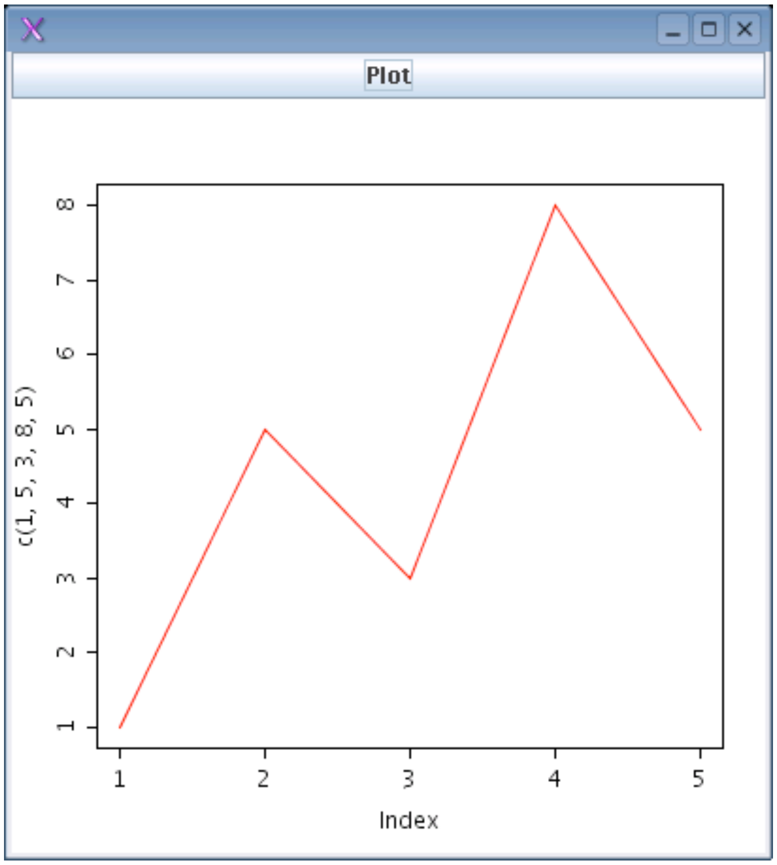
```
import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;

import org.rosuda.JRI.Rengine;
import org.rosuda.javaGD.GDCanvas;

public class JavaGDExample2 {

  /**
   * Create a Plottest JFrame and make it visible
   */
  public static void main(String[] args) {
    Plottest pf = new Plottest();
    pf.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
    pf.setVisible(true);
  }

  /**
```

```
   * The Plottest class. Creates an Rengine and makes it draw a simple plot
   * when the button is pressed.
   */
  public static class Plottest extends JFrame implements ActionListener {
    private static final long serialVersionUID = 1L;

    private Rengine _re;

    private JButton _plotB;

    // This is the GDCanvas that R will be plotting to. It has to be public
    // static for MyJavaGD2 to be able to access it.
    public static GDCanvas _gdc;

    public Plottest() {
      initR();
      initGUI();
    }

    /**
     * Create a plot button and the GDCanvas that R will be plotting to.
     */
    private void initGUI() {
      getContentPane().setLayout(new BorderLayout());
      _plotB = new JButton("Plot");
      _plotB.addActionListener(this);
      getContentPane().add(_plotB, BorderLayout.PAGE_START);
      _gdc = new GDCanvas(400, 400);
      getContentPane().add(_gdc, BorderLayout.PAGE_END);
      pack();
    }

    private void initR() {
      String[] dummyArgs = new String[1];
      dummyArgs[0] = "--vanilla";
      _re = new Rengine(dummyArgs, false, null);
      _re.eval("library(JavaGD)");
      _re.eval("Sys.putenv('JAVAGD_CLASS_NAME'='MyJavaGD2')");
    }

    /**
     * If the plot button is pressed, tell R to open a new JavaGD device and
     * to make a simple plot.
     */
    public void actionPerformed(ActionEvent e) {
      if (e.getSource() == _plotB) {
        _re.eval("JavaGD()");
        _re.eval("plot(c(1,5,3,8,5), type='l', col=2)");
        // R always draws a plot of a default size to the JavaGD device.
        // But our GDCanvas is supposed to have a different size, so
        // we have to resize it back to the size we want it to have.
        _gdc.setSize(new Dimension(400, 400));
        _gdc.initRefresh();
      }
    }

  }
}
```

File: MyJavaGD2.java

```
import org.rosuda.javaGD.GDInterface;

public class MyJavaGD2 extends GDInterface {

  /**
   * This is nice, huh? We don't need to create any frames or anything. Just
   * set c to the GDCanvas that we want R to plot to (in this case the GDCanvas
   * from the Plottest JFrame), and we're good!
   */
  public void gdOpen(double w, double h) {
    c = JavaGDExample2.Plottest._gdc;
  }

}
```

After running and compiling this, and pressing the "plot" button, you should get something like this:



Good bye, and thanks for all the fish

Well, I hope this has been at least a bit of a help for you. If there are any suggestions/questions, don't hesitate to mail me at falka777 (at) gmx.de. Also, remember that there's a wonderfully helpful stats-rosuda-devel mailing list at Uni Augsburg. Thanks for great support in finding out this stuff go to Simon from this list.

Have fun,

-Wojtek