A Java/JNI/C/Fortran makefile project for a Java plug-in and related Android app in Eclipse ADT bundle: A side-by-side comparison

R. de Beer and D. van Ormondt
Applied Physics, TU Delft, NL
E-mail: r.debeer@tudelft.nl

***Abstract*—We have developed a Java/Fortran based application, called MonteCarlo, that enables the users can carry out Monte Carlo studies in the field of *in vivo* MRS. The application is supposed to be used as a tool for the *j*MRUI platform, being the *in vivo* MRS software system of the TRANSACT European Union project. The MonteCarlo application can be launched either as a *j*MRUI custom plug-in (on Windows/Linux computers) or as a standalone Android app (on mobile Android devices). Both the plug-in and Android app version were developed as a Java/JNI/C/Fortran makefile project. This could be done by using one and the same version of Eclipse (in Eclipse ADT bundle), the main difference between the plug-in and Android app being the code, required for creating the GUI.**

***Index Terms*—Java/JNI/C/Fortran makefile project, Java plug-in, Android app, Eclipse ADT bundle, Java Swing vs Android user-interface model, *In vivo* MRS quantification, *j*MRUI software system, Monte Carlo modeling, Windows/Linux/Android OS, mobile devices**

I. Introduction

The purpose of this work was to investigate, whether a Java/Fortran based software application, created for computers with the Windows/Linux operating system, can be adapted for being used on mobile devices equipped with Android. We have done this on the hand of an example in the field of *in vivo* MRS [1]. It concerns an application, called MonteCarlo, that we have developed for performing Monte Carlo modeling [2].

We have created two versions of the MonteCarlo application, one being a Java custom plug-in for the *j*MRUI platform [3] (the *in vivo* MRS software system, supported as part of the TRANSACT European Union project [4]) and the other being a standalone Android app.

II. Methods

A. Design goals for the MonteCarlo application

We adopted the following design goals for the MonteCarlo application:

1) The application must be able to generate a large number of "*in vivo* MRS related" signals, all being obtained by adding to the "same noiseless simulated" MRS signal a different noise realization with the same standard deviation.
2) The noiseless simulated MRS signal must have been quantified by the *j*MRUI QUEST [3] quantification method, in this way yielding a QUEST-based $*$.results file that can act as input file for the MonteCarlo application.
3) The application must deliver a table with the Monte Carlo results of all noised signals.
4) The numerical computational part of the MonteCarlo application should be written in Fortran, since this programming language can handle double-precision complex numbers in a natural way (the MRS signals are complex-valued).

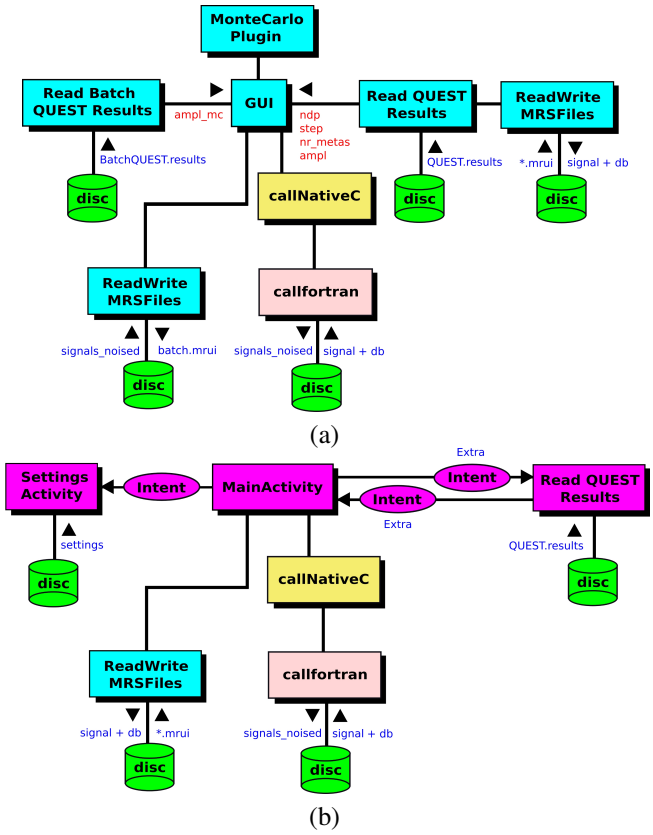


Figure 1: Conceptual design diagram of the main building blocks of the MonteCarlo (a) *j*MRUI plug-in and (b) Android app.

A consequence of design goal 2) is, that the MonteCarlo computer code must have a link to the *j*MRUI code. This means, since the core of *j*MRUI has been written in Java, that also the MonteCarlo application must have a Java part. Combined with design goal 4) we have chosen to use the Java/JNI approach. Since Fortran cannot be accessed directly from Java using JNI, we have applied ANSI C as intermediate language. That is to say, we have chosen to deal with a Java/JNI/C/Fortran project.

The design goals have resulted into conceptional diagrams presented in Figure 1 . In these diagrams the cyan blocks represent standard Java classes. The magenta blocks also represent Java classes, this time, however, with specific Android user-interface properties via the mechanism of being a subclass of the Android `Activity` class [5]. The yellow and pink block represent the ANSI C intermediate code and Fortran code, respectively, as being accessed from Java via the JNI mechanism. Finally, the magenta ovals depict the Android `Intent` class, which represent pieces of information, that can be sent among Android activities or other major building blocks [5].

1) `File->New->Java Project->project name->Finish`
   for starting a new Java project.
2) `project name*->New->Convert to a C/C++ Project->C Project->Makefile Project`
   for adding C/C++ nature to the project and for choosing the Makefile approach to build the MonteCarlo JNI native library.
3) `project name*->New->Folder->folder name->Finish`
   for adding a `jni` and `lib` folder to the project folder structure.
4) `src*->New->Class->Package->package name->Name->class name`
   for adding the project Java source code files to the project `src` folder. All should have the same chosen Java package name.
5) `Project->Properties->Java Build Path->Libraries->Add JARs...`
   for adding the desired Java JAR libraries to the project's Java build path. To that end the desired JARs first have to copied from outside into the project `lib` folder. In this context the `mrui.jar` file, containing the Java classes of the *j*MRUI software package, is the most important one.
6) Create and develop/edit in the project `jni` folder the `callNativeC.c` and `callfortran.f90` source code files and the GNU `makefile`.

(a)

1) `File->New->Other...->Android->Android Application Project->Next->Application Name->Project Name->Package Name-> ..... ->Next->Create Activity-> Activity Name->Layout Name->Finish`
   for starting a new Android app project. This sequence of steps delivers, amongst other things, a template file for the `MainActivity.java` Java source code (in the project `src` folder) and templates for other specific files of the Android app project.
2) Same as 2a.
3) Same as 3a.
4) Same as 4a. However, the first Java source code filename was already chosen in step 1b, as well as the related project Java package name.
5) Same as 5a.
6) Create and develop/edit in the project `jni` folder the `callNativeC.c` and the `callfortran.f90` source codes. Furthermore, add/edit the `Android.mk` and `Application.mk` Android `makefile`'s and copy from outside the `libfftw.so` and `liblapack.so` Android-suited libraries. The latter two were obtained by compiling the fftw C-codes with the Android NDK C-compiler and the LAPACK (with BLAS) Fortran-codes with our Android `gfortran` compiler.
7) `Project->Properties->C/C++ Build*-> Build Command->ndk-build`
   for using the Android NDK build script.
8) `Project->Properties->C/C++ General-> Paths and Symbols->Includes`
   for adding the required C/C++ include-directories of the `android-ndk-r9` Android NDK.

(b)

Figure 2: Essential Eclipse setup steps for the (a) *j*MRUI plug-in and (b) Android app. Note that * indicates "right-click on".

B. Choosing an Integrated Development Environment (IDE)

When developing a Java-based application we considered it a good choice of using Eclipse as our Java IDE. An important additional aspect of choosing Eclipse Java IDE was, that it can be combined with C/C++ support. Given the fact, that we also needed the Android Software Development Kit (SDK) [5] [6] for developing the MonteCarlo Android app version, we finally arrived at working with the Eclipse ADT bundle [6] combined with the Android Native Development Kit (NDK) [7]. The Eclipse ADT bundle includes a version of Eclipse Java IDE together with all essential Android development components/tools. The Android NDK is required for providing C/C++ support for the development of the MonteCarlo Android app version.

The above-mentioned means, that we could develop the MonteCarlo plug-in and Android app by using one and the same Eclipse Java IDE. Furthermore, C/C++/Fortran support was realized by using either the C/C++/Fortran software installed on Windows/Linux or by using the Android NDK. Concerning the latter, since the official Android NDK does not come with a `gfortran` compiler we additionally had to compile a `gcc` Fortran cross-compiler for Android [8].

In order to realize the Linux-suited plug-in version and the Android app version of the MonteCarlo application, we worked on the Linux Ubuntu 12.04 platform. In that case we used the `adt-bundle-linux-86-20131030` Eclipse ADT bundle [6] combined with the `android-ndk-r9` Android NDK (with an `ndk-r9-fortran-patch`, at the time of our work only tested on Ubuntu 12.04 [8]).

The Windows-suited version of the MonteCarlo plug-in was developed on the Microsoft Windows 7 platform. We then worked with the `adt-bundle-windows-86-20140702` Eclipse ADT bundle [6]. For getting C/C++/Fortran support under Windows 7 we installed MinGW (Minimalist GNU for Windows) [9].

C. Setting up Eclipse in Eclipse ADT bundle

When working with Eclipse to develop a specific (Java/JNI based) project, one get easily overwhelmed by the numerous features/choices/options, that one can select from in Eclipse in order to arrive at a certain result. In this report we will restrict ourselves to mentioning only the essential Eclipse setup steps, used for realizing the MonteCarlo application. In Figure 2 these setup steps are presented for both the plug-in and Android app version.

D. Details of the MonteCarlo Source codes

1) The MonteCarlo GUI: The Graphical User Interface (GUI) of the MonteCarlo plug-in version was realized in the same way as the *j*MRUI Desktop, that is to say, in a programmatic way by writing specific Java code including the user-interface components of the Java Swing package [10]. The GUI of the Android app version, however, was created in a mixed way by using a declarative approach (via XML) combined the programmatic approach using Java. Concerning the Java part, this was not written by using Java Swing (*not supported* in the Android platform), but by using Android's own user-interface (`widget`) package [5] (Android's own user-interface model is claimed to be better suited for mobile devices).

```
package mrui.custom.montecarlo;
import javax.swing.*;
//.....
public class Gui extends JInternalFrame {
  //.....
  private JButton quit;
  //.....
  public Gui(MruiImpl mrui) {
    //.....
  }
  private JPanel getDisplay() {
    //.....
    quit = new JButton("Quit");
    quit.setForeground(Color.red);
    quit.setBackground(Color.yellow);
    quit.addActionListener (new ActionListener () {
      public void actionPerformed (ActionEvent evt) {
        quitActionPerformed (evt);
      }
    }
    );
    //.....
    southpanel.add(quit);
    //.....
    mainpanel.add(southpanel, BorderLayout.SOUTH);
    //.....
    return mainpanel;
  }
  //.....
  private void quitActionPerformed (ActionEvent evt) {
    dispose();
  }
  //.....
}
```

(a)

```
<RelativeLayout xmlns:android=
  "http://schemas.android.com/apk/res/android"
  ......
  <LinearLayout
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginTop="270dp"
    android:layout_marginLeft="5dp" >
    <Button
      android:id="@+id/button_quit"
      android:layout_width="wrap_content"
      android:layout_height="30dp"
      android:background="#ffd398"
      android:textColor="#ff0000"
      android:text="Quit" />
  </LinearLayout>
  .....
</RelativeLayout>

package mrui.custom.montecarlo;
import android.widget.Button;
//.....
public class MainActivity extends Activity {
  //.....
  @Override
  protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    //.....
    Button quit = (Button)findViewById(R.id.button_quit);
    quit.setOnClickListener(
      new Button.OnClickListener() {
        public void onClick(View v) {
          finish();
        }
      }
    );
  }
  //.....
}
```

(b)

Figure 3: User-interface models for (a) the MonteCarlo plug-in (file `Gui.java`) and (b) the MonteCarlo Android app (files `activity_main.xml` and `MainActivity.java`). Shown are pieces of Java and XML code, used for realizing the `Quit` button of the MonteCarlo GUI (see also Figure 4).

To illustrate the two user-interface approaches, just mentioned, we present in Figure 3 pieces of Java and XML code, required for realizing a `Quit` button in the GUI of the plug-in and Android app version (see also Figure 4). Note, that the positioning of the button component in the Android app GUI is completely determined by the declarations in the XML code (via the `Layout`'s and `Margin`'s). Also its size, colors and text is determined in that way. This separation into XML and related Android Java gives freedom to change the presentation of an Android app GUI without disrupting its underlying functionality.

2) The MonteCarlo makefiles: When considering the steps 2 and 6 of Figure 2 (a) and (b), it becomes clear that we have chosen to build the MonteCarlo native libraries (`libmontecarlo.dll/.so` for Windows/Linux and `libmontecarlo.so` for Android) via the Makefile approach [11]. Since the contents of a makefile is essential for creating the proper native library, we display, as an example, in Figure 5 the makefile for the Windows plug-in and the Android app.

When comparing the two makefiles, we like to remark:

For Windows

1) The Windows `path`'s to Java, *j*MRUI and MinGW, installed on the local Windows computer, are explicitly present in the makefile.
2) Note the \'s in the Windows `path`'s, but also the /'s in the target rules.
3) Note the presence of `-Wl,--add-stdcall-alias`, required to overcome undefined symbols during the building of the library.
4) The required Java/JNI-related include file `mrui_custom_montecarlo_Gui.h` is generated via the makefile.
5) The Windows-suited Fortran libraries `liblapack.dll` and `libblas.dll` should be present in the MonteCarlo-project `jni` folder.

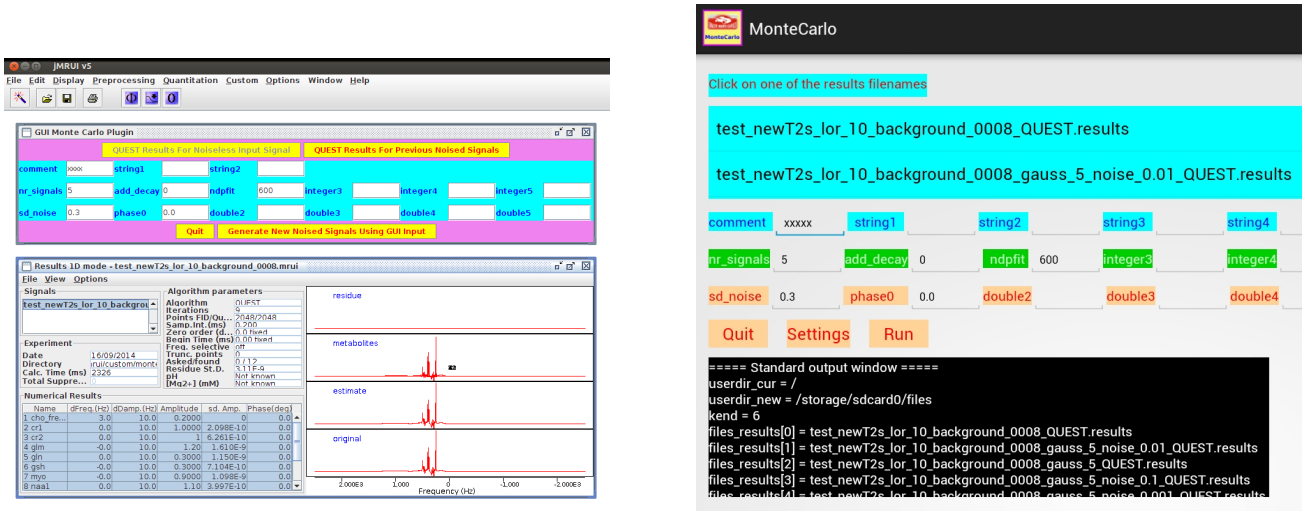
For Android

1) The only `path` information is about `LOCAL_PATH`, which in our case refers to the MonteCarlo-project `jni` folder.
2) In the `jni` folder two prebuilt Android-suited libraries should be present, called `libfftw.so` and `liblapack.so` (see also Figure 2 6) (b)).
3) The required Java/JNI-related include file `mrui_custom_montecarlo_MainActivity.h` is not realized via the makefile, but was generated outside the Eclipse MonteCarlo project (with the Java executable `javah`, using the Java class, concerned, and the proper Java Package Naming approach).

III. Building, installing and running the MonteCarlo application

A. Building MonteCarlo

In order to arrive at the moment of installing the MonteCarlo application as a *j*MRUI plug-in on Windows/Linux or as an Android



(a) (b)

Figure 4: (a) GUI of the MonteCarlo *j*MRUI plug-in, as displayed via Ubuntu 12.04. (b) GUI of the MonteCarlo Android app, as displayed via Android 4.1.2 on a Samsung GALAXY Note 8.0 tablet. Also displayed, in (a), is a selected *j*MRUI QUEST ∗.results file for getting input-values.

```
PATH_JAVA = C:\Program Files (x86)\Java\jdk1.6.0_27

PATH_JMRUI = C:\Users\beer\Documents\jmrui_5.0_matlab\
jMrui_v5.0_build_219_matlab\lib

all: libmontecarlo.dll copylib

libmontecarlo.dll: callfortran.dll mrui_custom_
montecarlo_Gui.h
  "C:\MinGW\bin\mingw32-gcc" -Wl,--add-stdcall-alias
  -I "$(PATH_JAVA)\include" -I "$(PATH_JAVA)\include\win32"
  -shared -lgcc -lm callfortran.dll -o libmontecarlo.dll
  callNativeC.c

callfortran.dll:
  "C:\MinGW\bin\mingw32-gfortran" -shared liblapack.dll
  libblas.dll -lm -lgfortran -o callfortran.dll
  callfortran.f90

mrui_custom_montecarlo_Gui.h: ../bin/mrui/custom/
montecarlo/Gui.class
  "C:\Program Files (x86)\Java\jdk1.6.0_27\bin\javah"
  -jni -classpath ../bin mrui.custom.montecarlo.Gui

copylib:
  cp liblapack.dll libblas.dll callfortran.dll
  libmontecarlo.dll $(PATH_JMRUI)
```

(a)

```
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)

LOCAL_MODULE := fftw-prebuilt
LOCAL_SRC_FILES := libfftw.so
LOCAL_EXPORT_C_INCLUDES := $(LOCAL_PATH)/include

include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE := lapack-prebuilt
LOCAL_SRC_FILES := liblapack.so

include $(PREBUILT_SHARED_LIBRARY)

include $(CLEAR_VARS)

LOCAL_MODULE        := montecarlo
LOCAL_SRC_FILES     := callNativeC.c callfortran.f90
LOCAL_C_INCLUDES    := $(LOCAL_PATH)/include
LOCAL_LDLIBS        := -llog -lgfortran
LOCAL_SHARED_LIBRARIES := fftw-prebuilt
LOCAL_SHARED_LIBRARIES := lapack-prebuilt

include $(BUILD_SHARED_LIBRARY)
```

(b)

Figure 5: Makefiles of the MonteCarlo application. (a) For the Windows plug-in version (called `makefile`). (b) For the Android app version (called `Android.mk`).

app on Android, one first has to build the MonteCarlo application, that is to say, to compile its ∗.java source code files and to generate its MonteCarlo native library. Building the Java classes in Eclipse is the easy part, because by default Eclipse is in the `auto-build` mode (taking care of compiling the ∗.java files automatically every time you change a Java code).

Generating the MonteCarlo native library means running one of the MonteCarlo makefiles (shown in Figure 5). Within Eclipse this is accomplished by carrying out the steps `makefile name*->Make Targets->Build->select Target->Build` . For the plug-in the selected target is called `all` (see Figure 5 (a)) with a corresponding build command `make` and for the Android app the target is called `montecarlo` (see Figure 5 (b)) with a build command `ndk-build`.

B. *Installing MonteCarlo*

Installing the MonteCarlo plug-in on Windows/Linux or the app on Android is different in the sense that on Windows/Linux the plug-in is added as a *new feature* to the already existing *j*MRUI application, whereas on Android the app is added as a *new standalone* application. This difference becomes clear, when considering the two installation procedures, as is shown in Figure 6.

C. *Running MonteCarlo*

The MonteCarlo plug-in version is launched via the Desktop Custom menu of the *j*MRUI system. Furthermore, launching the standalone Android app is accomplished by clicking its icon on the screen of the mobile device (but see also Figure 6 5) (b)). When running the application, there are differences between the plug-in and the Android app. They are related to the lack of the Java Swing

1) Copy the native library `libmontecarlo.dll/.so` to the *j*MRUI `lib` folder on the local computer. This is realized at the end of the Windows/Linux makefile (see Figure 5 (a) for the Windows example).
2) Copy a `MonteCarloPlugin.jar` file to the *j*MRUI `plugins` folder. This is accomplished by performing the steps `File->Export...->Java->JAR file ->Next->select resources-> path-to-jmrui-plugins-folder\ MonteCarloPlugin.jar->Finish`. The JAR file contains the MonteCarlo Java classes, as well as the required `montecarloplugin.properties` resource.

(a)

1) Enable in the mobile Android device, you want to install the MonteCarlo app on, the setting `Settings-> Developer options->USB debugging`.
2) Connect the mobile device (via USB) to the development computer (in our case with Ubuntu 12.04).
3) Install the app on the device by selecting `Run->Run` from the Eclipse menu bar.
4) If you run the app for the first time as an `Android Application`, the Android ADT will create a run configuration with an automatic target mode for device selection.
5) When performing step 3), a device chooser is presented showing the name of the device. After selecting the device, the app is installed and "run upon it".

(b)

Figure 6: Installing with Eclipse the MonteCarlo application on (a) a Windows/Linux computer (as a *j*MRUI plug-in) and (b) a mobile Android device (as an Android app).

1) "Before" launching the *j*MRUI plug-in for the first time, a desired working directory should be set by selecting the `Options->Setup options->Working Dir` via the *j*MRUI Desktop menu bar.
2) The first step, to be done in the MonteCarlo GUI, is to click the `QUEST Results For Noiseless Input Signal` button. After that, select the desired QUEST-based $*$.results file.
3) Minimize the $*$.results file. Note, that the GUI now shows the contents of the previous MonteCarlo session.
4) After changing/keeping the various GUI input fields, one can choose to click the `Generate New Noised Signals Using GUI Input` button. Note now, however, that the GUI is also enabled for clicking the `QUEST Results For Previous Noised Signals` button.

(a)

1) "After" launching the Android app for the first time, a desired working directory should be set via clicking the app `Settings` button.
2) The first step, to be done in the MonteCarlo GUI, is to select the desired QUEST-based input $*$.results file via clicking in a directory list.
3) Details of the selected $*$.results file are shown in a standard output window. Return to the main MonteCarlo GUI by clicking the `Return` button. Note, that the GUI now shows the contents of the previous MonteCarlo session.
4) After changing/keeping the various GUI input fields, one can click the `Run` button.

(b)

Figure 7: The various steps, required for running the MonteCarlo (a) plug-in and (b) Android app.

package support on the Android platform (see again subsubsection II-D1). Because of this lack of support the contents of the *j*MRUI $*$.results files can not be viewed in the GUI of the Android app. The various steps for running the two versions of the MonteCarlo application are presented in Figure 7.

IV. Example of a Monte Carlo result

An example of the results of a Monte Carlo study with the MonteCarlo application concerns a simulated *in vivo* MRS signal, with metabolite amplitudes (concentrations) related to the human brain [12]). The number of noised Monte Carlo signals, used in this study, was 1000. The purpose of the Monte Carlo study was to find out, whether or not denoising of the noised signals (with a wavelet approach [13]) may help to improve the quantification results (see Figure 8).

V. Brief discussion

A. Java Swing vs Android user interface

When searching on the Internet with key words like "java swing vs android user interface", one finds numerous links to webpages about comparing Java Swing and the Android user-interface model and about "how to modify" Java Swing applications for using on Android devices (see for instance [14], [15] and [16]). As far as we know, the conclusions in most articles/blogs usually come down to "rewriting the whole GUI-part", which in case of the *j*MRUI software system (with many Java-Swing based codes) is an almost impossible task. For the Android app version of our MonteCarlo application it meant, that we had to exclude using all graphical-presentation related code of the *j*MRUI QUEST $*$.results file.

Seen in the light of the existence of many important Java Swing applications, we agree with others, that the Android developers should consider to add full Java Swing support to the Android platform.

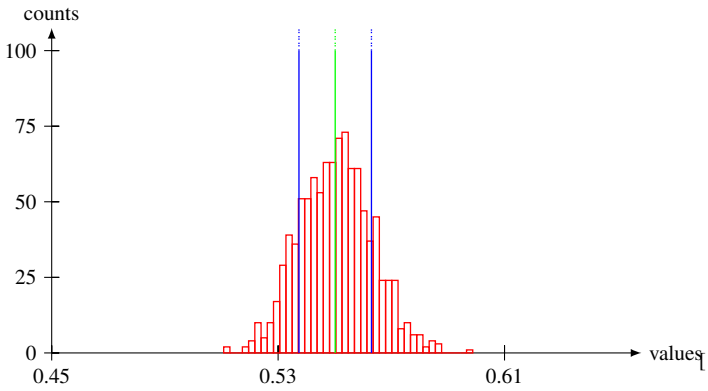
B. Checking Maximum Likelihood

When a Monte Carlo study with the MonteCarlo application is done, while obeying the "parametric" condition, it means that some Maximum Likelihood properties can be checked. Specifically, whether the estimated parameters are "unbiased" and whether their variances are "somewhat larger" than their related Cramér-Rao bounds (CRBs) [17].

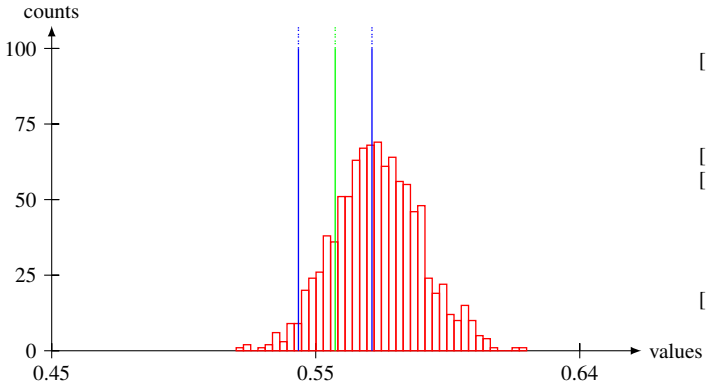
VI. Summarizing conclusions

Summarizing we like to make the following concluding remarks:

1) We have developed a Java/Fortran based application, called MonteCarlo, that enables the users to perform Monte Carlo studies in the field of *in vivo* MRS.
2) The MonteCarlo application is intended to be used as a tool for the *j*MRUI software package [3]. This can be done either as a *j*MRUI custom plug-in (on Windows/Linux computers) or as a standalone Android app (on mobile Android devices).
3) The MonteCarlo application could be developed as a Java/JNI/C/Fortran makefile project, using one and the same Eclipse Java IDE (in Eclipse ADT bundle [6]) for both the *j*MRUI plug-in and Android app.
4) When creating the MonteCarlo GUI, we worked with the "standard" Java Swing user-interface components [10] for the plug-in. For the Android app, however, we had to work with Android's own user-interface components [5] (due to the lack of Java Swing support on Android).
5) Seen in the light of item 4), porting large Java-Swing based software packages like *j*MRUI to Android seems to be an almost impossible task.
6) The MonteCarlo application offers the opportunity of investigating topics like "parametric vs semi-parametric", "Maximum Likelihood" properties and "bias-variance trade-off" [12] [17] [18].



(a)



(b)

Figure 8: Monte Carlo study of a simulated *in vivo* MRS signal, related to the human brain. Histogram of the Monte Carlo amplitudes of the myo-inositol metabolite (a) before denoising and (b) after denoising (with a wavelet approach). The green vertical line indicates the "true" amplitude value. Moreover, the blue vertical lines indicate the "true" value $\pm$ CRB. Note the bias in case (b), indicating "semi-parametric" estimation.

ACKNOWLEDGEMENT

This work was done in the context of FP7 - PEOPLE Marie Curie Initial Training Network Project PITN-GA-2012-316679-TRANSACT [4].

REFERENCES

[1] WikipediA, the free encyclopedia, "In vivo MRS," http://en.wikipedia.org/wiki/In_vivo_magnetic_resonance_spectroscopy, 2014, *In vivo* (that is 'in the living organism') magnetic resonance spectroscopy (MRS) is a specialised technique associated with magnetic resonance imaging (MRI). 1

[2] ——, "Monte Carlo method," http://en.wikipedia.org/wiki/Monte_Carlo_method, 2014, Monte Carlo methods are a broad class of computational algorithms that rely on repeated random sampling to obtain numerical results. 1

[3] D. Stefan, F. D. Cesare, A. Andrasescu, E. Popa, A. Lazariev, E. Vescovo, O. Strbak, S. Williams, Z. Starcuk, M. Cabanas, D. van Ormondt, and D. Graveron-Demilly, "Quantitation of magnetic resonance spectroscopy signals: the jMRUI software package," *Meas. Sci. Technol.*, vol. 20, p. 104035 (9pp), 2009. 1, 5

[4] TRANSACT European Union project, "Welcome to Transact!" http://www.transact-itn.eu/, 2013. 1, 6

[5] developer.android.com, "Introduction to Android," http://developer.android.com/guide/index.html, 2014, Android provides a rich application framework that allows you to build innovative apps and games for mobile devices in a Java language environment. 1, 2, 5

[6] ——, "Get the Android SDK," http://developer.android.com/sdk/index.html, 2014, . 2, 5

[7] ——, "Android NDK," https://developer.android.com/tools/sdk/ndk/index.html, 2014, The NDK is a toolset that allows you to implement parts of your app using native-code languages such as C and C++. 2

[8] Danilo Giulianelli, "Danilo's Tech Blog," http://danilogiulianelli.blogspot.nl/2013/02/how-to-build-gcc-fortran-cross-compiler.html, 2013, How to build the gcc Fortran cross-compiler for Android (ARM and x86) . 2

[9] MinGW.org, "MinGW," http://www.mingw.org/, 2014, MinGW, a contraction of "Minimalist GNU for Windows", is a minimalist development environment for native Microsoft Windows applications) . 2

[10] WikipediA, the free encyclopedia, "Swing (Java)," http://en.wikipedia.org/wiki/Swing_(Java), 2014, Swing is the primary Java GUI widget toolkit. 2, 5

[11] ——, "Makefile," http://en.wikipedia.org/wiki/Makefile, 2014, A Makefile is executed with the make command. 3

[12] D. van Ormondt, R. de Beer, J.W.C. van der Veen, D.M. Sima and D. Graveron-Demilly, "Error-Bars in Semi-Parametric Estimation," in *Proceedings ICT.OPEN 2013*. Van der Valk Hotel Eindhoven, The Netherlands: NWO/STW, 27-28 November 2013, pp. 15–20. 5

[13] Unpublished results. 5

[14] WikipediA, the free encyclopedia, "Comparison of Java and Android API," http://en.wikipedia.org/wiki/Comparison_of_Java_and_Android_API, 2014, This article compares the Java and Android API and virtual machines. 5

[15] Patrick Decker, "Writing and Styling Android Applications Coming from Swing," http://www.centigrade.de/en/blog/article/writing-and-styling-android-applications-coming-from-swing/, 2010, A Java developer who is used to developing GUIs with Swing and who is now trying to get into Android might be surprised: Java is not the same on Android. 5

[16] Stack Overflow, "Swing-Library for Android?" http://stackoverflow.com/questions/16383173/swing-library-for-android, 2013, Stack Overflow is a question and answer site for professional and enthusiast programmers. It's 100% free, no registration required. 5

[17] WikipediA, the free encyclopedia, "Estimation theory," http://en.wikipedia.org/wiki/Estimation_theory, 2014, Estimation theory is a branch of statistics that deals with estimating the values of parameters based on measured/empirical data that has a random component. 5

[18] D. van Ormondt, R. de Beer, J.W.C. van der Veen, D.M. Sima, and D. Graveron-Demilly, "Bias-Variance Trade-Off in In Vivo Metabolite Quantitation," in *Proceedings ICT.OPEN 2012*. WTC Rotterdam, The Netherlands: NWO/STW, 22-23 Qctober 2012. 5