Report on behalf of the EU-funded TRANSACT project

Monte Carlo Modeling for *in vivo* MRS: Generating and quantifying simulations via the Windows, Linux and Android platform

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

2014-11-10 14:24

Abstract

We have developed a Java/JNI/C/Fortran based software application, called MonteCarlo, with which 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 supporting 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 devices with the Android platform). Both the MonteCarlo plug-in and the Android app version were developed as a Java/JNI/C/Fortran Makefile project by using one and the same version of the Eclipse Java IDE, the main difference between the two MonteCarlo versions being the codes, required for creating the GUI. We have tested the two versions of the MonteCarlo application with a few Monte Carlo studies, which enabled us to verify specific topics of *in vivo* MRS Monte Carlo modeling, such as "parametric vs semi-parametric" estimation, checking "Maximum Likelihood" properties and dealing with the "Bias-Variance trade-off" problem.

Index Terms

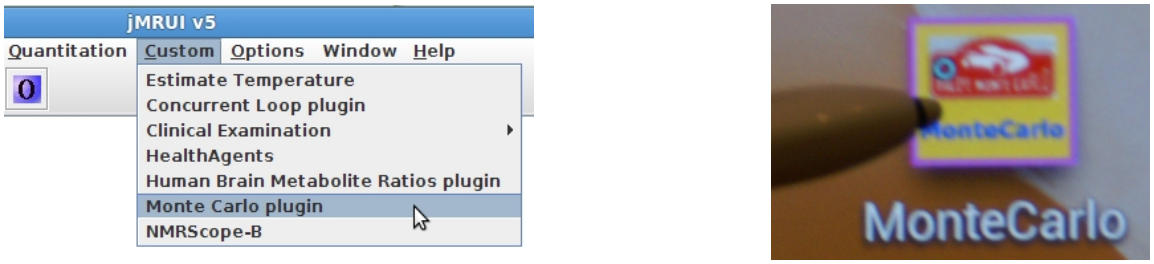
In vivo MRS quantification, Monte Carlo modeling, batch simulations, noised signals, *j*MRUI custom plug-in, Android app, Windows/Linux/Android OS, mobile devices, Java/JNI/C/Fortran Makefile project, Eclipse ADT bundle, Java Swing vs Android user-interface model, parametric vs semi-parametric, Maximum Likelihood, Bias-Variance trade-off

CONTENTS

I. Introduction

We have developed a Java/JNI application [1], called MonteCarlo, that is capable of performing Monte Carlo studies [2] in the field of *in vivo* Magnetic Resonance Spectroscopy (MRS) [3]. This work was done in the context of providing support for the *j*MRUI plug-in platform [4] [5], being the *in vivo* MRS software package of the TRANSACT European Union project [6].

We have created two versions of the MonteCarlo application, one being a custom plug-in that can be launched via the *j*MRUI software package on computers equipped with the Microsoft Windows or Linux platform, and one being a standalone Android app, that can be launched on mobile devices powered with the Android platform (see Figure 1).



(a) (b)

Figure 1: Example of launching the MonteCarlo application by using (a) the *j*MRUI Desktop Custom menu for the plug-in version and (b) icon clicking with a Samsung Galaxy S-Pen for the Android app version.

In section II we describe and compare details of the development of the plug-in version for Windows/Linux and of the app version for Android. Next, in section III we describe, how the MonteCarlo application can be built and installed on Microsoft Windows/Linux and on Android and we give a kind of User's Guide description of how to run the two versions. Next, in section IV we present some numerical examples of Monte Carlo results and in section V we briefly discuss a few aspects of the MonteCarlo application. Finally, in section VI we summarize the conclusions of this work.

II. Methods

A. Design goals for the MonteCarlo application

When starting to develop the MonteCarlo application, we adopted the following design goals:

1) The application must be able of generating a large number (say of the order of 1000) 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 [5] 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 a batch quantification 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).

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 [1] 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 (one for the plug-in and one for the Android app) as presented in Figure 2 . In these diagrams the cyan blocks represent standard Java classes. The magenta blocks also represent Java classes, this time, however, with having specific Android (user interface) properties via the mechanism of being a subclass of the Android `Activity` class [7] [8]. 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 (data and/or actions), that can be sent among Android activities or other major building blocks of Android apps [7] [8].

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 [9]. An important additional aspect of choosing Eclipse Java IDE was, that it can be combined with C/C++ support. Given the fact,

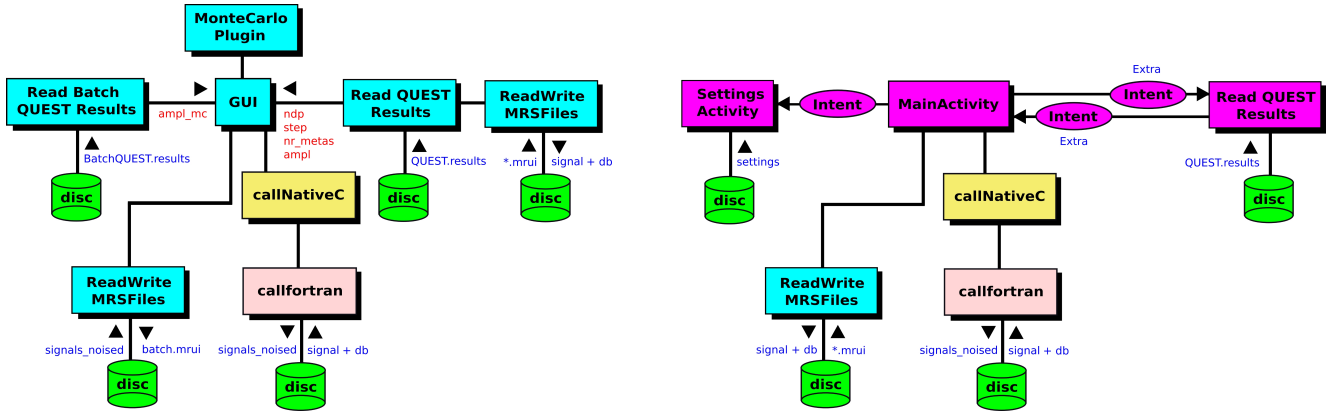


Figure 2: Conceptual design diagram of the main building blocks of (a) the MonteCarlo *j*MRUI plug-in and (b) the MonteCarlo Android app.

that we also needed the Android Software Development Kit (SDK) [7] [8] [10] for developing the MonteCarlo Android app version, we finally arrived at working with the Eclipse ADT Bundle [10] combined with the Android Native Development Kit NDK [11]. 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 [12].

In order to realize the Linux plug-in and Android app version of MonteCarlo we worked on the Ubuntu 12.04 platform. In that case we used the `adt-bundle-linux-86-20131030` Eclipse ADT Bundle [10] 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 [12]).

The Windows 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 [10]. For getting C/C++/Fortran support under Windows 7 we installed MinGW (Minimalist GNU for Windows) [13].

C. Setting up Eclipse in Eclipse ADT Bundle for realizing the MonteCarlo application

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 most important Eclipse setup steps, used for realizing the MonteCarlo application. In Figure 3 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 [14]. 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 [8]. 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 [7] (Android's own user-interface model is claimed to be better suited for mobile devices).

To illustrate the two user-interface approaches, just mentioned, we present in Figure 4 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 5). 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.

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 [8].
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 3: The most important Eclipse setup steps for (a) the MonteCarlo *j*MRUI plug-in and (b) the MonteCarlo Android app. Note, that * indicates "right-click on".

2) The MonteCarlo makefiles: When considering steps 2 and 6 of Figure 3 (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 [15]. Since the contents of a makefile is essential for creating the proper native library, we display, as an example, in Figure 6 the makefile for the Windows plug-in and Android app version.

When comparing the two makefiles, we like to make the following remarks:

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 3 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).

```
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 4: 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 5).

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 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).

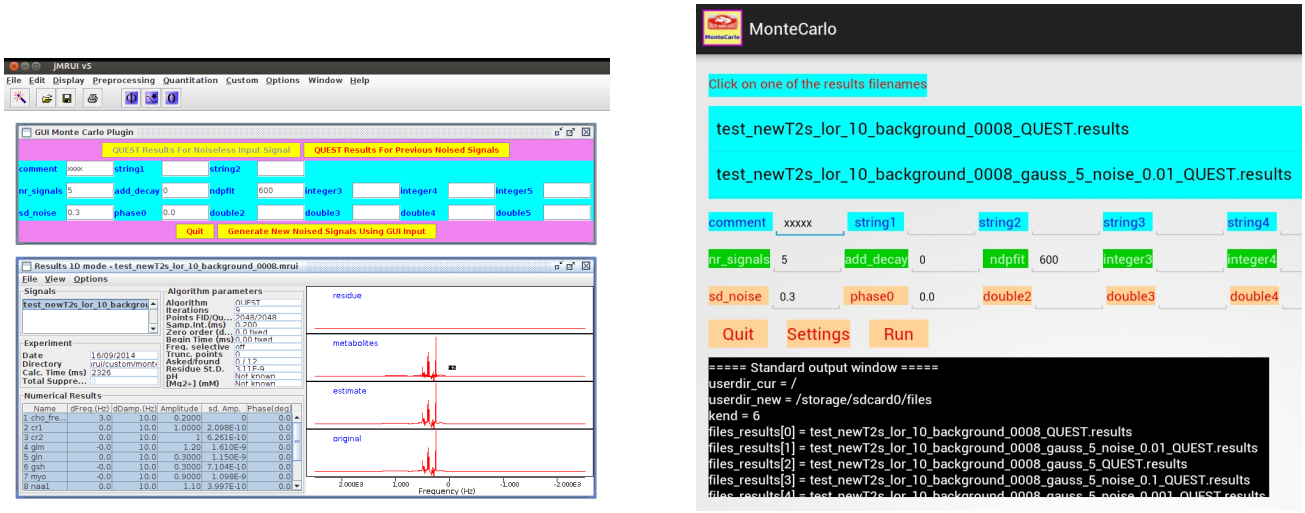
Generating the MonteCarlo native library means running one of the MonteCarlo makefiles (shown in Figure 6). Within Eclipse this is accomplished by carrying out the steps `makefile name*->Make Targets->Build->select Target->Build`. For the plug-in the selected target is called `all` (see Figure 6 (a)) with a corresponding build command `make` and for the Android app the target is called `montecarlo` (see Figure 6 (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 7.

C. Running MonteCarlo

The MonteCarlo application can be launched, as shown in Figure 1 (but see also Figure 7 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 package support on the Android platform (see again subsubsection II-D1). Because of this lack of support the contents of the



(a) (b)

Figure 5: (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 6: Makefiles of the MonteCarlo application. (a) For the Windows plug-in version (called `makefile`). (b) For the Android app version (called `Android.mk`).

*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 8.

Remarks, related to the steps in Figure 8 :

1) The working directory should contain the desired input files and a subdirectory, called `signals_noised`. The input files are a `gui_info.txt` text file (with info about the plug-in GUI-contents of the previous plug-in session), the desired QUEST-based *j*MRUI ∗.results file, the related simulated, noiseless, *in vivo* MRS ∗.mrui signal file and the contributing metabolite ∗.mrui signal files.
2) The *j*MRUI ∗.results file, selected, should be suited in the sense, that the residue is zero (that is to say, that the amplitudes, frequencies, dampings and phases have the "true" values).
3) The contents of the `gui_info.txt` text file is only changed after clicking the `Generate New Noised Signals Using GUI Input` button (or the `Run` button in the Android app GUI).
4) • After clicking the `Generate New Noised Signals Using GUI Input` button (or the `Run` button in the

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 6 (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 7: 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 8: The various steps, to be done for running the MonteCarlo (a) plug-in and (b) Android app.

Android app GUI), all noised signals for the MonteCarlo study are generated and stored on disk. Furthermore, all signals are quantified in the Fortran code using a Gauss-Newton based fitting algorithm.

- By clicking the `QUEST Results For Previous Noised Signals` button, the plug-in is triggered to show the QUEST quantification results for a previous MonteCarlo session. This QUEST quantification should have been carried out in *j*MRUI "outside" the MonteCarlo plug-in.

IV. Results

In this section we present the results of performing some *in vivo* MRS Monte Carlo studies with the MonteCarlo application. The first study concerns a simulated *in vivo* MRS signal, derived from a real-world signal, measured in the human brain at a static magnetic field strength of 3T by using the General Electric (GE) provided PRESS measurement protocol [16] [17] . The simulated metabolite basis set signals. used for constructing the simulated noiseless input signal for the MonteCarlo application (see above about the *j*MRUI $*$.results input file), were calculated by applying the GAMMA NMR C++ library [18] (thereby including details of the GE PRESS measurement protocol and the 3T MRS scanner, concerned). Also, in the metabolite basis set signals the effects of the transverse relaxation (signal loss and Lorentzian decay) were taken into account.

The input parameters of the MonteCarlo GUI, used in this study (see also Figure 5 (b)), were `comment` = xxxxx (this comment is attached to some of the output files), `nr_signals` = 5 (number of noised signals in the Monte Carlo study), `add_decay` = 0 (no extra decay function added), `ndpfit` = 600 (number of data points in the quantification), `sd_noise` = 0.3 (standard deviation of the noise) and `phase0` = 0 (zero-order phase in degrees). The simulated basis set consisted of 11 metabolite signals and one background signal, with amplitudes, related to the real-world GE signal. These amplitudes were taken relatively with respect to that of the creatine metabolite (with values derived from human brain *in vivo* MRS [19]). Note, that in this output example `nr_signals` was taken only as small as 5. In actual Monte Carlo studies this number should be much larger.

File Edit View Search Tools Documents Help

montecarlo_fitexp_xxxxx.txt

comment: xxxxx
Results for amplitudes (a.u.)

nr	true	mean	stdev	bias
1	0.2000	0.2005	0.0015	-0.0005
2	1.0000	1.0005	0.0188	-0.0005
3	1.0000	1.0080	0.0528	-0.0080
4	1.2000	1.2028	0.0599	-0.0028
5	0.3000	0.2964	0.0336	0.0036
6	0.3000	0.3221	0.0153	-0.0221
7	0.9000	0.9060	0.0797	-0.0060
8	1.1000	1.1315	0.0445	-0.0315
9	1.1000	1.1379	0.0408	-0.0379
10	0.2000	0.1718	0.0316	0.0282
11	0.0300	0.0248	0.0067	0.0052
12	1.0000	1.0089	0.0325	-0.0089

(a)

(b)

File Edit View Search Tools Documents Help

montecarlo_amplitudes_quest_xxxxx.txt

Monte Carlo results for amplitudes (a.u.)

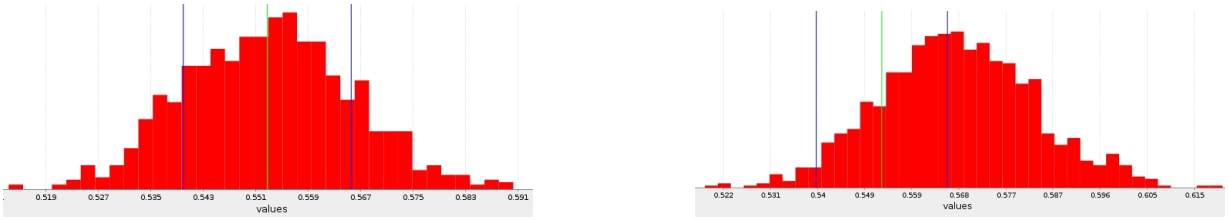
metabolite name	true	mean	stdev	bias	crb	stdev_crb
cho_freqshift_-3	0.2000	0.1998	0.0014	0.0002	0.0039	0.0000
cr1	1.0000	1.0008	0.0164	-0.0008	0.0144	0.0001
cr2	1.0000	1.0060	0.0464	-0.0060	0.0373	0.0002
glm	1.2000	1.1972	0.0761	0.0028	0.0615	0.0005
gln	0.3000	0.3048	0.0350	-0.0048	0.0505	0.0008
gsh	0.3000	0.3233	0.0198	-0.0233	0.0334	0.0004
myo	0.9000	0.9077	0.0708	-0.0077	0.0630	0.0004
naa1	1.1000	1.1077	0.0212	-0.0077	0.0178	0.0005
naa2	1.1000	1.1102	0.0499	-0.0102	0.0713	0.0004
naag	0.2000	0.1926	0.0144	0.0074	0.0146	0.0005
scyllo	0.0300	0.0241	0.0063	0.0059	0.0059	0.0000
xbackground	1.0000	1.0034	0.0291	-0.0034	0.0341	0.0001

(c)

Figure 9: MonteCarlo based metabolite amplitudes, as determined with a Gauss-Newton based quantification for (a) the plug-in and (b) the Android app and as determined with the *j*MRUI QUEST method for (c) the plug-in. For details of the MonteCarlo study see text.

Tables (a) and (b) of Figure 9 show, that the numerical results of the Gauss-Newton quantification are exactly the same for the MonteCarlo plug-in and the Android app (as expected). Furthermore, table (c) of Figure 9 shows, that the columns `mean`, `stdev` and `bias` of the *j*MRUI QUEST results differ slightly from those of the Gauss-Newton results. The latter probably is due to a better numerical stability of QUEST for low SNR, as can be verified by "decreasing" the value of the `sd_noise` GUI input.

A second example of the results of a Monte Carlo study with the MonteCarlo application (now only with the plug-in version) concerns a simulated *in vivo* MRS signal, again with metabolite amplitudes (concentrations) related to the human brain, but now supposed to be measured at a static magnetic field strength of 11.7T [20] . Apart from the much higher magnetic field strength, when compared to the first example above, another important difference is the much higher `nr_signals` (is now 1000). The purpose of the Monte Carlo study was to find out, whether or not denoising of the noised MonteCarlo signals (with a wavelet approach [21]) may help to improve the quantification results.



(a) (b)

Figure 10: Monte Carlo study (with the MonteCarlo plug-in version) of a simulated *in vivo* MRS signal, related to the human brain and supposed to be measured at a static magnetic field strength of 11.7T. Histogram of the amplitudes of myo-inositol (a) before denoising and (b) after denoising. The green vertical line indicates the "true" amplitude value. Moreover, the blue vertical lines indicate the "true" value $\pm$ CRB. For details of the Monte Carlo study see text.

Figure 10 shows a histogram of the Monte-Carlo-study produced amplitudes of the myo-inositol metabolite. In Figure 10 (a) the result corresponds to the "parametric" situation [20], that is to say, the simulated Monte Carlo signals are fitted with the correct model function and the noise has a Gaussian distribution. In Figure 10 (b), however, the result corresponds to the "semi-parametric" situation, which was found to be due to the denoising operation. In the latter case this semi-parametric situation can be concluded from the presence of the bias.

The histograms, just shown, are produced in the Java part of the Montecarlo plug-in version, by feeding the `HistogramChart` object of the `umontreal.iro.lecuyer.charts` package (see website of the SSJ library [22]) with the metabolite amplitudes, produced by the *j*MRUI QUEST quantification of all noised signals (see again Figure 8 4) (a)).

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 (see for instance [23], [24] and [25]). 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. Robustness of the Monte Carlo quantifications

In the quantification methods of the MonteCarlo application the non-linear least squares (NLLS) problem is solved by using either the Gauss-Newton algorithm (GNA) (in the subroutine `fitexp` of the Fortran code) or the Levenberg-Marquardt algorithm (LMA) (in the *j*MRUI QUEST method [26]). The LMA is known to be slower than the GNA but on the other hand it is "more robust", the latter meaning that it more often converges to a minimum [27] (not always the global minimum).

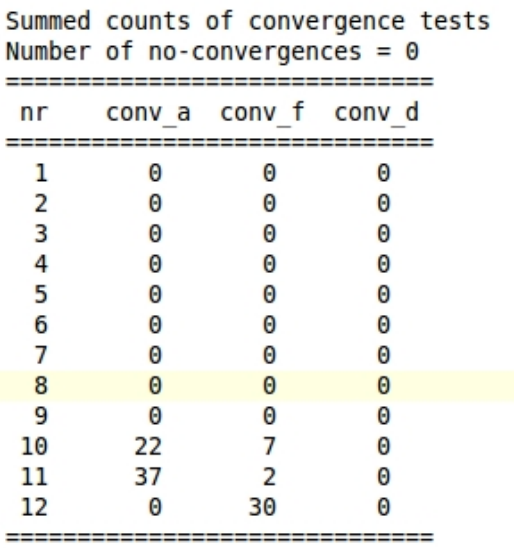
In order to follow the convergence of our `fitexp` method, we have created in the Fortran code the possibility of monitoring the parameter values (amplitudes, frequencies and dampings) after each NLLS iteration. We could not do this in the QUEST case, since our MonteCarlo plug-in version can only access the Java-based *j*MRUI core code.

If the relative change of a parameter with respect to its starting value is larger than a certain chosen value, this fact is counted and added to a "monitor table". This is done for all `nr_signals` noised signals. At the end of the Monte Carlo calculations the counts are added to a "total monitor table", which also contains the number of "no convergence" (counted, if a noised signal needs more than 100 `fitexp` iterations).

An example of such a total monitor table is shown in Figure 11 . It can be seen from it, that at the end of the Monte Carlo calculations the Gauss-Newton fit of all noised signals had reached convergence, that metabolite 10 and 11 had problems (during the NLLS iterations) with the amplitude and frequency convergence and that metabolite 12 (the simulated background) had problems with the frequency convergence.

C. 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 smaller" than their related Cramér-Rao bounds (CRBs) [28].



Summed counts of convergence tests
Number of no-convergences = 0

nr	conv_a	conv_f	conv_d
1	0	0	0
2	0	0	0
3	0	0	0
4	0	0	0
5	0	0	0
6	0	0	0
7	0	0	0
8	0	0	0
9	0	0	0
10	22	7	0
11	37	2	0
12	0	30	0

Figure 11: Total monitor table, belonging to the Monte Carlo study, presented in Figure 9 . For details see text.

Table (c) in Figure 9 shows, that for most of the metabolites the `bias` is significant smaller than the `stdev`. However, when comparing the `stdev`'s with the `crb`'s it can be seen that the second Maximum Likelihood property, mentioned above, is not really met (possibly due to the small value of `nr_signals` and/or the relatively large value of `sd_noise`). By increasing `nr_signals` and decreasing `sd_noise` (to avoid "threshold behavior") we could establish, however, that the Maximum Likelihood properties indeed were realized.

In the "semi-parameric" case the situation concerning reliable quantifications becomes much more complicated, because then the Maximum Likelihood properties are not valid and as a consequence one has to deal with the problem of the "Bias-Variance trade-off" [20] [29].

VI. Summarizing conclusions

Summarizing we like to make the following concluding remarks:

1) We have developed a Java/JNI/C/Fortran based application, called MonteCarlo, that enables the users to perform Monte Carlo studies in the field of *in vivo* MRS [3].
2) The MonteCarlo application is intended to be used as a tool for supporting the *j*MRUI software package [4] [5]. 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 application could be developed as a Java/JNI/C/Fortran Makefile project, using one and the same Eclipse Java IDE (being part of the Eclipse ADT Bundle [7] [8] [9] [10]) for both the *j*MRUI plug-in version and the Android app version.
4) When creating the MonteCarlo GUI, we worked with the "standard" Java Swing user-interface components [14] for the plug-in. For the Android app version, however, we had to work with Android's own user-interface components [7] (due to the lack of Java Swing support on Android).
5) Seen in the light of lacking full Java Swing support on Android, porting the *j*MRUI package to the Android platform will 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" [20] [28] [29].

Acknowledgement

This work was done in the context of FP7 - PEOPLE Marie Curie Initial Training Network Project PITN-GA-2012-316679-TRANSACT [6].

References

[1] WikipediA, the free encyclopedia, "Java Native Interface," http://en.wikipedia.org/wiki/Java_Native_Interface, 2014, JNI is a programming framework that allows Java code to call native applications. 2
[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. 2

[3] ——, "In vivo magnetic resonance spectroscopy," 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). 2, 10
[4] D. Stefan, A. Andrasecu, E. Popa, H. Rabeson, O. Strbak, Z. Starcuk, M. Cabanas, D. van Ormondt, and D. Graveron-Demilly, "jMRUI Version 4 : A Plug-in Platform," in *IEEE International Workshop on Imaging Systems and Techniques, IST 2008*, Chania, Greece, 10-12 September 2008, pp. 346–348. 2, 10
[5] 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. 2, 10
[6] TRANSACT European Union project, "Welcome to Transact!" http://www.transact-itn.eu/, 2013. 2, 10
[7] 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. 2, 3, 10
[8] M. Gargenta and M. Nakamura, *Learning Android (*2ND EDITION). O'Reilly, 2014. 2, 3, 4, 10
[9] The Eclipse Foundation, "Eclipse Is ..." https://www.eclipse.org/home/, 2014, An amazing open source community of Tools, Projects and Collaborative Working Groups. 2, 10
[10] developer.android.com, "Get the Android SDK," http://developer.android.com/sdk/index.html, 2014, . 3, 10
[11] ——, "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++. 3
[12] 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) . 3
[13] 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) . 3
[14] WikipediA, the free encyclopedia, "Swing (Java)," http://en.wikipedia.org/wiki/Swing_(Java), 2014, Swing is the primary Java GUI widget toolkit. 3, 10
[15] ——, "Makefile," http://en.wikipedia.org/wiki/Makefile, 2014, A Makefile is executed with the make command. 4
[16] J.W. van der Veen, J. Shen, D.van Ormondt and R. de Beer, "Quantifying In vivo 1H MRS in the human brain at 3T with simulated metabolite basis sets. Including details of the in vivo PRESS measurement protocol and MRS-scanner setup and performing MRS line shape analysis based on self-deconvolution ," 2013, Report on behalf of the MRS Core Facility, NIMH, NIH, USA. 7
[17] Jan Willem C. van der Veen, Ron de Beer, Dirk van Ormondt, Jun Shen, "Extraction of Glutamate from the GABA Edited Spectra," in *ISMRM 21st Annual Meeting and Exhibition*, Salt Lake City, Utah, USA, 20-26 April 2013, poster 2032. 7
[18] J.W. van der Veen, D. van Ormondt and R. de Beer, "Simulating Metabolite Basis Sets for *in vivo* MRS Quantification. Incorporating details of the PRESS Pulse Sequence by means of the GAMMA C++ library," in *Proceedings ICT.OPEN 2012*. WTC Rotterdam, The Netherlands: NWO/STW, 22-23 Qctober 2012, pp. 1–6. 7
[19] R. de Beer, "Human Brain Metabolite Ratios. Metabolite Transverse Relaxation Times," 2013, Custom plug-in for the *j*MRUI software package. Can also be used as standalone document. e-mail: r.debeer@tudelft.nl. 7
[20] 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. 8, 9, 10
[21] Unpublished results. 8
[22] Universit de Montral, "SSJ: Stochastic Simulation in Java," http://simul.iro.umontreal.ca/ssj/indexe.html, 2014, SSJ is a Java library for stochastic simulation, developed under the direction of Pierre L'Ecuyer, in the Dpartement d'Informatique et de Recherche Oprationnelle (DIRO). 9
[23] 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. 9
[24] 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. 9
[25] 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. 9
[26] Helene Ratiney, Yoeri Coenradie, Sophie Cavassila, Dirk van Ormondt and Danielle Graveron-Demilly, "Time-Domain Quantitation of 1H Short Echo-Time Signals: Background Accommodation," *Magn. Reson. Mater. Phys.*, vol. 16, pp. 284–296, 2004, . 9
[27] WikipediA, the free encyclopedia, "LevenbergMarquardt algorithm," http://en.wikipedia.org/wiki/LevenbergMarquardt_algorithm, 2014, In mathematics and computing, the LevenbergMarquardt algorithm (LMA) is used to solve non-linear least squares problems. 9
[28] ——, "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. 9, 10
[29] 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. 10