Short Tutorial for ODIN, the Object-oriented Development Interface for NMR

Thies H. Jochimsen

November 8, 2016

Contents

1 Sequence Programming

1.1 Introduction

In this step-by-step tutorial, we will discover how to program a magnetic resonance (MR) sequence with ODIN. As a representative example, a fast spin-echo (FSE) sequence (also termed turbo spin echo or RARE sequence) will be implemented in this tutorial to introduce ODIN's sequence programming interface and to highlight the benefits of using a truly object-oriented approach for sequence development. To fully appreciate this tutorial, you should be familiar with the concepts of object-oriented programming (OOP) (data encapsulation, inheritance, polymorphism), and in particular, with OOP programming in C++. Before going into detail, here is the full source code of the FSE sequence:

```
#include <odinseq/seqall.h>

class METHOD_CLASS : public SeqMethod {

  LDRint Shots;

  SeqPulsar exc;
  SeqPulsar refoc;
  SeqGradPhaseEnc pe1,pe2;
  SeqGradConstPulse spoiler;
  SeqAcqRead read;
  SeqAcqDeph deph;
  SeqObjList echopart;
  SeqObjList prep;
  SeqObjList oneshot;
  SeqDelay echodelay;
  SeqDelay prepdelay;
  SeqDelay relaxdelay;
  SeqObjLoop echoloop;
  SeqObjLoop shotloop;

 public:
  METHOD_CLASS(const STD_string& label) : SeqMethod(label) {}

  void method_pars_init() {

    Shots=4;
    append_parameter(Shots,"Shots");

  }

  void method_seq_init() {

    exc=SeqPulsarSinc("exc",geometryInfo->get_sliceThickness(),
                      true,2.0,commonPars->get_FlipAngle());

    refoc=exc;
    refoc.set_label("refoc");
    refoc.set_rephased(false);
    refoc.set_flipangle(180.0);
    refoc.set_phase(90.0);
  refoc.set_pulse_type(refocusing);


  pe1=SeqGradPhaseEnc("pe1",commonPars->get_MatrixSize(phaseDirection),
                      geometryInfo->get_FOV(phaseDirection),
                      phaseDirection,0.25*systemInfo->get_max_grad(),
                      centerOutEncoding,interleavedSegmented,Shots);

  pe2=pe1;
  pe2.set_label("pe2");
  pe2.invert_strength();


  spoiler=SeqGradConstPulse("spoiler",sliceDirection,
                            0.5*systemInfo->get_max_grad(),1.0);


  read=SeqAcqRead("read",commonPars->get_AcqSweepWidth(),
                  commonPars->get_MatrixSize(readDirection),
                  geometryInfo->get_FOV(readDirection),readDirection);

  deph=SeqAcqDeph("deph",read,spinEcho);


  echodelay.set_label("echodelay");

  echopart= refoc + spoiler + echodelay + pe1 + read + pe2 + echodelay + spoiler;


  prepdelay.set_label("prepdelay");

  prep= exc + prepdelay + spoiler/deph;


  relaxdelay.set_label("relaxdelay");

  oneshot= prep + echoloop( echopart )[pe1][pe2];

  set_sequence(
    shotloop( oneshot  + relaxdelay )[pe1.get_reorder_vector()][pe2.get_reorder_vector()]
  );
}


void method_rels() {

  float minTEprep = 2.0* (exc.get_duration() - exc.get_magnetic_center()
                    + (spoiler/deph).get_duration()
                    + refoc.get_magnetic_center());

  float minTEecho=echopart.get_duration();

  float minTE=minTEprep;
```

```
 95       if(minTEecho>minTE) minTE=minTEecho;
 96
 97       commonPars->set_EchoTime(minTE,noedit);
 98       prepdelay.set_duration(0.5*(minTE-minTEprep));
 99       echodelay.set_duration(0.5*(minTE-minTEecho));
100
101       if(oneshot.get_duration()>commonPars->get_RepetitionTime())
102         commonPars->set_RepetitionTime(oneshot.get_duration());
103       relaxdelay=commonPars->get_RepetitionTime()-oneshot.get_duration();
104     }
105
106
107     void method_pars_set() {
108       read.set_reco_vector(line,pe1);
109     }
110
111   };
112
113   ODINMETHOD_ENTRY_POINT
```

This code can be compiled by saving it to a file with extension .cpp and open it in the ODIN user interface by File→Open.

1.2 The MR sequence is a C++ class

Because each MR sequence is an entity with local data (sequence parameters, sequence objects) and functions to operate on this data (initialization, execution, simulation), it is obvious to bundle this structure in a C++ class. This is exactly how sequences are implemented in ODIN: You create a class that is derived from the base class of all sequences, `SeqMethod`. This is done in **line 3**. Please note that instead of using the term sequence, the term *method* is often used instead within ODIN to emphasize that the sequence as whole is addressed instead of single sequence objects.

Three remarks: First, the `#include` statement in **line 1**integrates all headers necessary for sequence programming. Second, the macro `METHOD_CLASS` must be used as the class label for every sequence.[1] Third, the macro `ODINMETHOD_ENTRY_POINT` in **line 113**expands to some extra code required by ODIN and must be at the end of each sequence file.

Next, we need a public constructor that accepts a string as a single argument and passes this string to `SeqMethod` as done in **line 23**. This constructor is necessary to be able to create an instance of the class while giving it a unique label which is used throughout the ODIN framework to refer to this sequence. The wicked `STD_string` is equivalent to `std::string` for internal reasons.[2]

Finally, to complete our skeleton of the sequence, we must implement a bunch of virtual functions as shown in **line 25,32,86,107**. The purpose of these functions is:

`method_pars_init`	Initialize sequence parameters
`method_seq_init`	Initialize sequence objects and create sequence layout
`method_rels`	(Re)calculate sequence timings
`method_pars_set`	Final preparations before executing the sequence

[1] It will be replaced by a unique string each time the sequence is compiled within the ODIN user interface so that each time it is linked into the process space, a unique symbol is available.

[2] Some ancient systems have none or a broken C++ standard library and no namespaces. Thus, the macros starting with `STD_` are used instead of `std::` so that ODIN can replace them by its own implementations, if necessary.

The ODIN framework will call these functions during initialization/preparation of the sequence. In order to realize our custom sequence, we have to implement these functions. Hence, in the remainder of this tutorial, we will fill these functions with the appropriate code to program our FSE sequence.

1.3 Sequence Parameters and Their Initialization

In the following, the term *sequence parameters* refers to properties that modify the layout, and hence, the resolution, timing, etc. of the sequence. Examples are the echo time (TE), the repetition time (TR) and the field of view (FOV). A number of pre-defined sequence parameters are available within the sequence class. They can be accessed via the following pointers and contain the specified class of parameters:

Pointer label:	Pointing to type:	Description:
`commonPars`	`SeqPars`	Common MR parameters (e.g. TE, TR)
`geometryInfo`	`Geometry`	Geometry parameters (e.g. FOV)
`systemInfo`	`System`	System settings (e.g. field strength)

Single parameters of these *parameter blocks* can be retrieved and modified with appropriate `get_` and `set_` methods, as we will see later. We will also refer to these global objects as *proxies* because the hide platform specific stuff behind a common interface. Please refer to the online manual for a list of available member functions.

If you write a custom sequence, you probably want to be able to define and use an extra set of parameters. As it turns out, this is pretty easy in ODIN, all you have to do is to declare the parameter as a (private) member of your method class and tell ODIN that this is a new sequence parameter. For example, we will need an extra parameter, `Shots`, in our FSE sequence for the number of excitations for each image. Thus, we first add the parameter as a member to our class (**line 5**). `LDRint` is the type of our new parameter and makes `Shots` behave exactly like a built-in `int`.[3] Next, we can assign a default value to the parameter in `method_pars_init` (**line 27**). To register the parameter so that it is recognized by ODIN, for example to display it in the graphical user interface (GUI), we must call the function `append_parameter` in `method_pars_init` of our sequence (**line 28**). After compiling the method, you will see a field with the new parameter on the right-hand side of the ODIN GUI. As we will see later, the parameter can be used exactly like a regular `int` in our method code.

1.4 Designing the Sequence

In this section, we will create all the sequence objects required for our FSE sequence. The term *sequence objects* refers to all basic elements of an MR sequence, such as RF pulses, gradient pulses and periods of data acquisitions. In ODIN, these sequence objects are data members of the sequence class. In addition, we will see how these sequence objects are combined into sequence containers and used together with sequence loops to form the whole sequence.

1.4.1 RF Pulses

As the FSE sequence is basically a Carr-Purcell-Meiboom-Gill (CPMG) sequence, we will need an excitation pulse and a refocusing pulse. The excitation pulse `exc` is declared in **line 7**. It is of type `SeqPulsar` because it uses the same functions as the Pulsar program to calculate pulses. The pulse is initialized in **lines 34-35**by using the constructor of the specialized pulse class `SeqPulsarSinc`[4]. Several parameters are passed to the constructor:

[3]There are many other types emulating built-in types, such as `LDRfloat` and `LDRdouble`. Composite types are also available, such as `LDRstring`, `LDRcomplex` and arrays. Please refer to the online manual for a complete list.

[4]The following goes on behind the scenes: First, a temporary (unnamed) object of type `SeqPulsarSinc` is created with the given parameters. Next, the assignment operator of `SeqPulsar` is called. Because `SeqPulsarSinc` is

The first assigns a label to the object created. It is convenient to use the same label as that used in source code. The label will be used to refer to the sequence object, for example when plotting the sequence. Next, the constructor expects the slice thickness which is simply taken from the global geometry proxy pointed to by `geometryInfo`. The third parameter is set to `true`, to automatically add a rephasing gradient to the excitation pulse. Finally, we specify the pulse duration (4th parameter) and the flip angle (5th parameter).

In an FSE sequence, it is usually a good idea to use the same pulse shape for excitation and refocusing, except for the flip angle. Thus, for the refocusing pulse `refoc` (declared in **line 8**) we make use of the assignment operator of `SeqPulsar` to create a copy of `exc` and change only those properties which differ (**lines 37-42**): First, the label is changed to `refoc`. Then, the rephasing gradient pulse is removed because it is not necessary for a refocusing pulse. Next, the phase is set to 90°(CPMG condition) and the flip angle is changed. Finally, the pulse type is set to `refocusing`[5].

1.4.2 Gradient Pulses

For phase encoding, i.e scanning of lines in k-space , two gradient pulses with variable strength are required to create a linear phase prior to data acquisition and to rewind this phase after acquisition. We could use the class `SeqGradVector` to achieve this, but then we would have to calculate the gradient strengths ourselves to cover a certain FOV for a certain resolution. Since we are lazy, and ODIN provides a convenient class `SeqGradPhaseEnc` to achieve just that, we will use it to declare both objects (**line 9**). The initialization of the first phase encoding gradient `pe1` in **lines 45-48**requires some explanation: After specifying the label, resolution (matrix size) and FOV, the channel of the gradient pulse is set to `phaseDirection`[6]. Next, the maximum gradient strength is set to 1/4 of that possible with the available gradient hardware. The duration of the gradient pulse will then be calculated to achieve the desired phase encoding. The argument `centerOutEncoding` specifies that instead of linear encoding (scanning k-space from one edge to the other in sequential order), the central lines are acquired first and then moving outwards from the center. Finally, the last two arguments are used to separate the phase encoding lines into `Shots` interleaves by using `interleavedSegmented` as the *reordering scheme*. We will see later how iterate through the phase encoding interleaves and shots by sequence loops. In **lines 50-52**, the phase rewinder `pe2` is initialized by making it a copy of `pe1` and inverting its polarity.

To suppress free-induction signal of the refocusing pulses, a pair of spoiler gradients is required around each refocusing pulse. This fairly simple constant gradient pulse of type `SeqGradConstPulse` is declared in **line 10**and initialized in **lines 55-56**with 1/2 of the maximum gradient strength and 1 ms duration.

1.4.3 Data Acquisition

To sample one line in k-space between two refocusing pulses in our FSE sequence, an acquisition period has to be created together with a simultaneous read gradient. We could create this composite object ourselves by using a simple acquisition of type `SeqAcq` and a trapezoidal gradient `SeqGradTrapez`, but again, ODIN provides a convenient composite class to achieve this, namely `SeqAcqRead` which is an acquisition together with a gradient pulse of the same duration. The object `read` of this type is declared in **line 11**and initialized in **lines 59-61**. As usual, the first argument to the constructor of `SeqAcqRead` is the object's label. Then, the sweep width (sampling frequency) is specified together with the matrix size in read direction (number of sampling points) and the desired FOV. The desired gradient

derived from `SeqPulsar`, all data members of `SeqPulsar` of the temporary object are copied over to `exc`, thereby initializing `exc`.

[5]This has no direct effect on the sequence time course, but makes calculation gradient moments (e.g. k-space analysis) possible in the plotting GUI.

[6]This is one of the three possible channels `readDirection`, `phaseDirection` and `sliceDirection` which correspond to the three orthogonal directions in the logical patient coordinate system, which usually differs from the coordinate system spanned by the three gradient coils if the imaging plane is tilted.

channel for the gradient is the `readDirection`.

This is a good place to explain the concept of *Interface base classes*, sometimes called *interface classes*: An Interface base class is a class which provides a set of (mostly virtual) member function. This class cannot be instantiated, i.e. no object of this class can be created, but it serves as a base class to concrete implementations of this set of member functions (the interface). For example the class `SeqAcqRead` implements two interfaces, namely `SeqAcqInterface`, which is the base class for all acquisition classes, and `SeqGradInterface`, which is the base class for all classes with one or more gradient pulses. In this way, all acquisition (or gradient) objects share a common interface. For instance, the member function `get_acquisition_start` is reimplemented in each acquisition class to return the duration from the start of the sequence object to the beginning of the actual acquisition (this is necessary for proper timing calculations, see below). The implementation of this function can be very different, depending on the concrete acquisition class used: `SeqAcq` will return the duration to trigger the digitizer, and `SeqAcqRead` will also consider the duration of the gradient ramp of the read gradient. However, the programming interface is the same in both cases which makes it easy to remember the member functions and, more importantly, allows *polymorphism*: If different acquisition types should be supported within one sequence, member functions can be called via a pointer (or reference) to type `SeqAcqInterface` instead of several `if`-statements.

In order to start sampling at one edge of k-space in read direction, we also need a pre-dephasing gradient pulse. Again, we could create this ourselves using a `SeqGradConstPulse` object, but there is a more convenient class `SeqAcqDeph` which can be employed. It is declared in **line 12**. After the obligatory label as the first argument to the constructor (**line 63**), a reference to an object derived from `SeqAcqInterface` is expected for which the dephaser will be created. As we have seen above, the object `read` is derived from `SeqAcqInterface` and can therefore be used as the second argument so that a suitable pre-dephaser will be created. The last argument `spinEcho` specifies that the pre-dephaser will have the same polarity as the read gradient which is suitable for spin-echo (and, therefore, CPMG) acquisition.

1.4.4 Sequence Containers and Delays

Having initialized all our basic sequence elements, we now start to group them together to build the whole MR sequence. For this purpose, *sequence containers* of type `SeqObjList` are used: For example, we will group the objects within one refocusing interval together in **line 68**and assign this composite object to `echopart` (declared in **line 13**) so that we can loop over this part of the sequence later. As you can see, sequence objects are concatenated by using the + operator.

However, just concatenating objects is not enough for our FSE sequence, we must also fulfill CPMG timing constraints. Therefore, the *sequence delay* `echodelay` of type `SeqDelay` (declared in **line 16**and initialized in **line 66**[7]) is used to insert certain waiting periods into the sequence. We will calculate the duration of this delay later together with the duration of other delay objects. For now, we just create the basic layout of the sequence.

It is also convenient to group the preparation phase into the object `prep` (**line 14**) as shown in **line 73**. Here, the delay `prepdelay` (**line 17**, **line 71**) will be used to achieve CPMG timing. Again, we use the + operator to concatenate sequence objects. But since `spoiler` and `deph` are on different gradient channels, they can be played out simultaneously. This is exactly what the / operator does: It makes sequence objects simultaneous, if possible. Otherwise, it will generate an error message.

[7]Although setting the label is not crucial, it helps debugging as this label shows up in all visualizations of the sequence.

1.4.5 Sequence Loops

Next, we will build the sequence part of one shot, that is, the preparation part `prep` followed by a loop over `echopart` (the sequence part of one echo). In ODIN, loops are possible with the class `SeqObjLoop`, which is, like `SeqObjList`, a container to hold other sequence objects in sequential order. In the following, this contents will be called the *kernel* of the loop. In addition, `SeqObjLoop` allows looping over the kernel to repeat its execution. At each iteration, properties of so-called *sequence vectors* can be changed. These sequence vectors are all classes derived from `SeqVector`. Thereby, a loop can be employed as follows: First, a loop object is declared in **line 19**. Then, to iterate over a kernel `kern` while iterating properties of `vec1`, `vec2`, ..., the basic syntax is[8]

```
echoloop ( kern ) [vec1][vec2]...
```

In our sequence, this syntax is used in **line 78**to iterate over `echopart` while iterating the values of `pe1` and `pe2`. Here iterating simply means to change the strength of the phase encoding gradient so that different lines in k-space are scanned at each echo, according to the encoding and reordering scheme specified above.

But we are not done yet because we have only build the sequence for one shot. Thus, we need another loop `shotloop` (**line 20**) to iterate over the different shots. At each of these iterations, `pe1` and `pe2` have to be aware that the next interleave of phase encoding steps has to be used. For this purpose, each sequence vector has a *reordering vector*. Whenever a certain reordering scheme is used (as done above by the constructor of `SeqGradPhaseEnc` or by the function `set_reorder_scheme`), the reordering vector is used to iterate over the set of modified arrangements of the values of the vector, for example over the interleaves, if `interleavedSegmented` is used as the reordering scheme. The reordering vector, just like every other vector, can be attached to a loop to iterate over it. It can be retrieved from its parent vector by the function `get_reorder_vector`. This is done in **lines 80-82**: The reordering vectors of `pe1` and `pe2` are attached to the loop `shotloop` while repeating `oneshot`. A padding delay `relaxdelay` (**line 18**, **line 76**) is also added which we use later to adjust the correct TR.

Within the same line, the function `set_sequence` of the base class `SeqMethod` is used to specify the whole sequence, i.e. to tell the ODIN framework what to use when plotting/executing/simulating our sequence. Using this function is usually the last step in `method_seq_init`.

1.5 (Re)Calculating Sequence Timings

In the previous section, we have created the basic layout of the sequence, but ignored the timing constraints of our FSE sequence. This will be done now, namely in the member function `method_rels` of our method class[9]. First, we will calculate the minimum TE possible which is equivalent to the CPMG refocusing interval. This either limited by the minimum duration possible with our preparation (**lines 88-90**), or by the refocusing interval (**line 92**). Here, the functions `get_duration` and `get_magnetic_center` return the duration of the corresponding sequence object[10] and the time point of the center of the pulse relative to its beginning, respectively. Next, we calculate the maximum of both in **lines 94-95**and set the global TE (in `commonars`) to this value (**line 97**). Here, the second argument `noedit` indicates that the parameter cannot be edited by the user, it will be indicated as such in the GUI. Finally, we adjust the duration of the padding delays to match the calculated TE in **line 98**and **line 99**using the `set_duration` function.

[8]This syntax becomes possible by overloading the ( ) and [ ] operator of `SeqObjLoop`.

[9]Using a separate member function for timing calculations, which are usually fast compared to all the initialization stuff in `method_seq_init`, has the advantage that, if only timing settings of the sequence change, `method_rels` is the only function to be called.

[10]The container `echopart` is a sequence object itself whereby `get_duration` returns the total duration of the concattenated objects it contains.

A similar technique is used to set the minimum TR possible and adjust the corresponding padding delay (**lines 101-103**).

1.6 Image Reconstruction

The sequence is now complete and could be executed on the scanner, but image reconstruction would fail because ODIN does not know where to put each line, which was acquired with the `read` object, in k-space . We have to inform ODIN explicitely that the vector object `pe1` was responsible for phase encoding. This is done in **line 108**. The first argument of `set_reco_vector` is of type `recoDim` which is an enum used to specify so-called *reconstruction dimensions*. These can be, for instance:

- Phase encoding step (`line`)
- Slice index (`slice`)
- Echo number (`echo`)
- User-defined index (`userdef`)

The second argument is a vector object which was responsible to iterate through this dimension. For example, if we would add multi-slice excitation, we would add a frequency list to `exc` (using `set_freqlist`) and then use `exc` as the slice vector.

After execution of the sequence, the complex data of the acquired NMR signal is stored on disk in the so-called scan directory (scandir) in the order it was acquired. This is achieved by using the native mechanisms (e.g. on Bruker and GE) or by a custom reconstruction routine which does nothing but dumping the raw data to the hard drive (e.g. on Siemens). However, just storing the raw data would not be enough to reconstruct an image. Extra information like the one above (location of each acquisition in k-space) is required. Therefore, an extra file `recoInfo`, which contains this information in human-readable ASCII format, is stored together with the raw data in the scandir. The file is a serialized version of the class `RecoPars`, which means that on object of this type can read/write the file and queried subsequently for its values.

Finally, the program `odinreco`, which is part of ODIN will retrieve all information from `recoInfo` and generate image data. It is executed in the scandir on the command line with

```
odinreco
```

to create the reconstructed image data. Please refer to the online manual for further documentation how to fine tune the reconstruction.

Index

List of Abbreviations