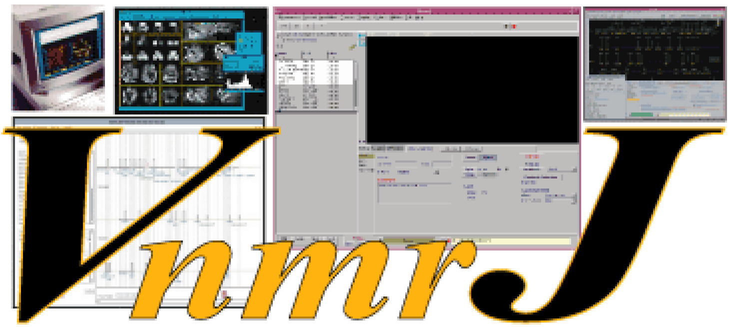

Pulse Sequence Looping

VARIAN

Paul Kinchesh
European MRI Applications Laboratory
Oxford, UK
September 2008

Introduction

The exact order that a sequence loops depends on a number of factors:

1. How sequence loops are ordered within the pulse sequence
2. The setting of the `seqcon` parameter.
3. The setting of the `array` parameter.
4. The setting of `il` and `bs` parameters.

Pulse Sequence Looping Functions

The following sequence functions are available for looping in a pulse sequence:

Function	**Description**
`loop`	A "compressed" loop
`msloop`	A multiple slice loop, a "compressed" or "standard" loop
`peloop`	A phase encode loop, a "compressed" or "standard" loop
`peloop2`	A 2nd phase encode loop, a "compressed" or "standard" loop
`nwloop`	A "compressed" loop for use in `NOWAIT` gradients

Compressed Loops, Standard Loops and Parameter Arrays

Compressed Loops
Compressed loops are executed as many times as defined by the loop control arguments in ***each*** pass through the pulse sequence.

Standard Loops
Standard loops execute ***multiple*** passes through the pulse sequence. The number of passes is defined by the loop control arguments.

Parameter Arrays
Parameter arrays cause ***multiple*** passes through the pulse sequence in the same way as a standard loop. The number of passes is determined by the number of elements in the parameter array.

Data acquisition and nf

All of the data that are acquired in a single pass through the pulse sequence are stored within the same data block of the raw `fid` file.

The raw `fid` file contains as many data blocks as there are passes through the experiment.

The parameter `nf` must be set to the number of traces* to be acquired in each block of data. The `setloop` macro is used to set `nf` appropriately.

*A trace is typically a single FID or echo, i.e. a single acquisition of np points of data.

Parameter Arrays

By default, when parameters are created, they are set with protection bit 8 (value 256) off.
⟶ Multiple values of a parameter cause an acquisition array
⟶ the `array` parameter is set to include the name of the arrayed parameter
⟶ the `arraydim` parameter is set to the size of the arrayed experiment.
⟶ counter `ix` is used internally in pulse sequences to index the current array element with $1 \leq ix \leq arraydim$

Example.

The default gems (2D gradient echo multi slice) parameters and sequence has compressed muti-slice and phase encode loops to give a 2D multi-slice data set

```
nf=ns*nv               /* There are ns×nv traces in each data block */
array=''               /* array is an empty string */
arraydim=1             /* The sequence will execute once */

te=0.005,0.01,0.015    /* Set three echo times */
⟶ array='te'           /* The array string includes te */
⟶ arraydim=3           /* The sequence will execute 3 times, once with each te */
```

Multiple Parameter Arrays

`array='x,y'` indicates the parameters x and y are arrayed, with y taking precedence.
⟶ order of experiments is x_1y_1, x_1y_2,... x_1y_n, x_2y_1, x_2y_2,...x_2y_n,... x_my_n
⟶ `arraydim=m×n`

`array='y,x'` indicates the parameters x and y are arrayed, with x taking precedence.
⟶ order of experiments is x_1y_1, x_2y_1,... x_my_1, x_1y_2, x_2y_2,...x_my_2,... x_my_n
⟶ `arraydim=m×n`

`array='(x,y)'` indicates the parameters x and y are jointly arrayed.
⟶ order of experiments is x_1y_1, x_2y_2,... x_ny_n
⟶ `arraydim=n`

Joint arrays can have up to 10 parameters.

Regular multiple arrays can have up to 20 parameters, with each parameter being either a simple parameter or a diagonal array.

The total number of elements in all arrays can be 2^{32}–1.

The last parameter in the array string cycles the fastest

seqcon

The `secqon` parameter is a string of five characters.

Each character in the string corresponds to data dimensions which are defined as follows.

seqcon character index		Dimension	Parameters
VnmrJ Parameter	**Pulse sequence**		
`[1]`	`[0]`	Multi-echo	`ne`
`[2]`	`[1]`	Multi-slice	`ns,pss`
`[3]`	`[2]`	1st phase encode	`nv,ni`
`[4]`	`[3]`	2nd Phase encode	`nv2,ni2`
`[5]`	`[4]`	3rd Phase encode	`nv3,ni3`

Each character must take one of the following values:

- `'c'` to signify a compressed loop
- `'s'` for a standard loop
- `'n'` for no loop

The `secqon` parameter is used to the set multi-slice and phase encode loops to be either compressed, standard or not present.

`secqon` is used in pulse sequences with `msloop`, `peloop` and `peloop2` functions.

`secqon` is evaluated by the `setloop` macro to calculate `nf`.

The loop function

The function `loop` starts a compressed loop of the statements between the `loop` and `endloop` functions. Real-time variables are used to control the number of times that the loop is executed.

Example.

```
#include "sgl.c"

pulsesequence()
{
  int vne     = v1;     /* Real-time variable for number of echoes */
  int vne_ctr = v2;     /* Real-time variable for echo loop counter */

  init_mri();           /* Retrieve standard sequence parameters including
                           ne, the number of echoes in a multi-echo sequence */

  initval(ne,vne);      /* Initialize vne */
  loop(vne,vne_ctr);    /* Start loop */

    /* Pulse sequence statements for gradient prescription and acquisition go here */

  endloop(vne_ctr);     /* End loop */
}
```

loop(vne,vne_ctr);

The real-time variable `vne` is used to specify the number of times control is to pass through the loop. `vne` can be any positive number or zero.

The real-time variable `vne_ctr` is used as a counter to track of the number of times control has passed through the loop. At the first pass through the loop `vne_ctr` is zero.

endloop(vne_ctr);

The `endloop` function checks `vne_ctr` against `vne` to figure whether to pass control back to the `loop` function or to following pulse sequence statements.

seqcon

Multi-echo sequences typically use the compressed loop function.

The 1st character of the `seqcon` string should be set to `'c'` for a compressed loop.
The `setloop` macro then calculates `nf` to include ne passes through the compressed loop.

If the 1st character of the `seqcon` string is set to `'n'` the `setloop` macro sets `ne=1` and then calculates `nf` accordingly.

If the 1st character of the `seqcon` string is set to `'s'` the `setloop` macro reports an error.

The msloop function

The function `msloop` starts a multi-slice loop to execute statements between the `msloop` and `endmsloop` functions. Real-time variables are used to control the loop.

Example.

```
#include "sgl.c"

pulsesequence()
{
  int vns     = v1;     /* Real-time variable for number of slices */
  int vns_ctr = v2;     /* Real-time variable for slice loop counter */

  init_mri();           /* Retrieve standard sequence parameters including
                           ns, the number of slices in a multi-slice sequence */

  msloop(seqcon[1],ns,vns,vns_ctr);     /* Start multi-slice loop */

    /* Pulse sequence statements including acquisition go here */

  endmsloop(seqcon[1],vns_ctr);         /* End multi-slice loop */
}
```

seqcon, ns and pss

If a `msloop` is used the 2nd character of the `seqcon` string can be set to:
`'c'` for a compressed multi-slice loop or `'s'` for a standard multi-slice loop.

The `pss` parameter is an array of positions of slices in a multi-slice experiment.

If the 2nd character of the `seqcon` string is set to `'c'` for a compressed multi-slice loop:
⟶ `setprotect('pss','on',256)` is set so `pss` is not an acquisition array.
⟶ `pss` is removed from the `array` parameter string (if it is there).
⟶ `arraydim` is updated accordingly.
⟶ `setloop` macro sets `ns` to the size of the `pss` array.
⟶ `setloop` macro calculates `nf` to include `ns` passes.

If the 2nd character of the `seqcon` string is set to `'s'` for a standard multi-slice loop:
⟶ `setprotect('pss','off',256)` is set so `pss` is an acquisition array.
⟶ `pss` is added to the `array` parameter string.
⟶ `arraydim` is updated accordingly.
⟶ `setloop` macro sets `ns=1`.
⟶ `setloop` macro calculates `nf` to include `ns` passes.

Multiple Parameter Arrays

If the 2nd character of the `seqcon` string is set to `'s'` for a standard multi-slice loop
→ `pss` is added to the `array` parameter string

If other parameters are also arrayed
→ the precedence with which `pss` cycles is determined by its position in the `array` string.

msloop(seqcon[1],ns,vns,vns_ctr);

The `msloop` function is defined in `/vnmr/psg/rtcontrol.c`
The behaviour depends on the value of the character passed in the 1st argument.

In the example, `seqcon[1]` is passed - the **2nd** character of the `seqcon` string.

If `seqcon[1]='c'` then:
If `ns`≥`1` then `initval(ns,vns);` else `initval(1.0,vns);`
A compressed loop is then executed using `loop(vns,vns_ctr);`

If `seqcon[1]='s'` then:
If `ns>1` the sequence aborts.
Otherwise the following are set: `initval(ns,vns); initval(0,vns_ctr);`

Any other value of `seqcon[1]` causes the sequence to abort.

endmsloop(seqcon[1],vns_ctr);

The `endmsloop` function is defined in `/vnmr/psg/rtcontrol.c`
The behaviour depends on the value of the character passed in the 1st argument.

If `seqcon[1]='c'` then `endloop(vns_ctr);` is used to control the compressed loop.

Any other value of `seqcon[1]` just returns control to the sequence.

The peloop function

The function `peloop` starts a phase encode loop to execute statements between the `peloop` and `endpeloop` functions. Real-time variables are used to control the loop.

Example.

```
#include "sgl.c"

pulsesequence()
{
  int vnv     = v1;    /* Real-time variable for number of phase encodes */
  int vnv_ctr = v2;    /* Real-time variable for phase encode loop counter */

  init_mri();          /* Retrieve standard sequence parameters including
                          nv, the number of phase encode steps in a sequence */

  peloop(seqcon[2],nv,vnv,vnv_ctr);    /* Start phase encode loop */

    /* Pulse sequence statements including acquisition go here */

  endpeloop(seqcon[2],vnv_ctr);        /* End phase encode loop */
}
```

seqcon, nv, nv2, nv3, ni, ni2 and ni3

`peloop` and `peloop2` work in a way that is often misunderstood (not surprisingly) …

If a `peloop` is used the 3rd character of the `seqcon` string can be set to:
`'c'` for a compressed multi-slice loop or `'s'` for a standard multi-slice loop.

Parameters are set as follows

seqcon character	`'c'`	`'s'`	`'n'`
1st	`ne`	`error`	`ne=1`
2nd	`ns=size('pss')`	`ns=1`	`ns=1`
3rd	`nv,ni=1`	`nv,ni=nv`	`nv=0,ni=1`
4th	`nv2,ni2=1`	`nv2,ni2=nv2`	`nv2=0,ni2=1`
5th	`nv3,ni3=1`	`nv3,ni3=nv3`	`nv3=0,ni3=1`

`ne`, `nv`, `nv2`, `nv3` on their own mean the values simply remain as they have been set

nv: number of views in 1st phase encode dimension
nv2: number of views in 2nd phase encode dimension
nv3: number of views in 3rd phase encode dimension

seqcon, nv, nv2, nv3, ni, ni2 and ni3

The use of `ni`, `ni2` and `ni3` is borrowed from high resolution NMR

`ni`: number of increments of the evolution time `d2` in 2nd indirectly detected dimension
`ni2`: number of increments of the evolution time `d3` in 3rd indirectly detected dimension
`ni3`: number of increments of the evolution time `d4` in 4th indirectly detected dimension

"Hidden" arrays of size `ni`, `ni2` and `ni3` are set for those values that are > 1

→ `array='x,y'` with `ni>1`, `ni2>1` and `ni3>1`, can be thought of as
`array='n3,n2,n,x,y'`
where `n`, `n2`, and `n3` are arrays of size `ni`, `ni2`, and `ni3` respectively.

This also demonstrates how the evolution times are incremented with respect to each other and any other array elements.

→ The evolution times are incremented more slowly than all other array elements.

→ `d4` is incremented more slowly than `d3` which is incremented more slowly than `d2`.

This is exactly how standard phase encode loops are set.

NB The "hidden" arrays of size `ni`, `ni2` and `ni3` do show up in the pop-up array window.

What is the current element?

For standard phase encode loops we need to know exactly what phase encode step to prescribe for any given array element.

Counter `ix` is used internally in pulse sequences to index the current array element with
`1` ≤ `ix` ≤ `arraydim`

Integers `d2_index`, `d3_index`, and `d4_index` are calculated in integer math to provide the indices of the respective arrays according to

```
d2_index = ((ix-1)/(arraydim/(ni*ni2*ni3))) % ni

d3_index = ((ix-1)/(arraydim/(ni2*ni3))) % ni2

d4_index = (ix-1)/(arraydim/ni3)
```

% is the modulus operator that computes the remainder of integer division.

peloop, seqcon, nf and arraydim

When the value of the `seqcon` string is changed the `setloop` macro is run to ensure that the correct values of `nf`, `ni`, `ni2` and `ni3` are set with respect to the specified `seqcon`.
`arraydim` is updated internally with a VnmrJ command `calcdim`.

peloop(seqcon[2],nv,vnv,vnv_ctr);

The `peloop` function is defined in `/vnmr/psg/rtcontrol.c`
The behaviour depends on the value of the character passed in the 1st argument.

In the example, `seqcon[2]` is passed - the **3rd** character of the `seqcon` string.

If `seqcon[2]='c'` then:
If nv≥1 then `initval(nv,vnv);` else `initval(1.0,vnv);`
A compressed loop is then executed using `loop(vnv,vnv_ctr);`

If `seqcon[2]='s'` then:
If nv≥1 then: `initval(nv,vnv); initval((double)d2_index,vnv_ctr);`
Otherwise: `assign(zero,vnv); assign(zero,vnv_ctr);`

Any other value of `seqcon[2]` causes the sequence to abort.

endpeloop(seqcon[2],vnv_ctr);

The `endpeloop` function is defined in `/vnmr/psg/rtcontrol.c`
The behaviour depends on the value of the character passed in the 1st argument.

If `seqcon[2]='c'` then `endloop(vnv_ctr);` is used to control the compressed loop.

Any other value of `seqcon[2]` just returns control to the sequence.

The peloop2 function

The function `peloop2` starts a phase encode loop to execute statements between the `peloop2` and `endpeloop` functions. Real-time variables are used to control the loop.

Example.

```
#include "sgl.c"

pulsesequence()
{
  int vnv2     = v1;    /* Real-time variable for number of PE steps */
  int vnv2_ctr = v2;    /* Real-time variable for phase encode loop counter */

  init_mri();         /* Retrieve standard sequence parameters including nv2,
                         the number of PE steps in 2nd phase encode dimension */

  peloop2(seqcon[3],nv2,vnv2,vnv2_ctr);    /* Start phase encode loop */

    /* Pulse sequence statements including acquisition go here */

  endpeloop(seqcon[3],vnv2_ctr);           /* End phase encode loop */
}
```

peloop2(seqcon[3],nv2,vnv2,vnv2_ctr);

The `peloop2` function is defined in `/vnmr/psg/rtcontrol.c`
The behaviour depends on the value of the character passed in the 1st argument.

In the example, `seqcon[3]` is passed - the **4th** character of the `seqcon` string.

The `peloop2` function is used to allow prescription of standard phase encode loops in both the 1st and 2nd phase encode dimensions. Otherwise the `peloop` function could be used twice in, for example, a 3D sequence with two phase encode dimensions.

It's ***only*** the use of the `d3_index` that makes the `peloop2` function different:

If `seqcon[3]='s'` then:
If `nv2`≥1 then: `initval(nv2,vnv2); initval(`**`(double)d3_index`**`,vnv2_ctr);`
Otherwise: `assign(zero,vnv2); assign(zero,vnv2_ctr);`

The nwloop function

The `nwloop` function is used to loop during a gradient prescribed with a `NOWAIT` flag.

Pulse sequence functions for prescribing gradients have a "wait" flag that can either be

`WAIT` the pulse sequence waits for the gradient to be played out before continuing.

`NOWAIT` the pulse sequence continues whilst the gradient is played out.

When a `NOWAIT` gradient is used the total time duration of a loop that is executed during the gradient must be known at run time so that the sequence can be properly interpreted.

The standard compressed `loop` function only takes real-time variables as arguments.
The function `nwloop` takes an additional argument (double) for the number of loops so that it can be used with a `NOWAIT` gradient.

Example.

```
#include "sgl.c"

pulsesequence()
{
  double nloops,duration,level;

  int vnw = v1;        /* Real-time variable for nowait loop */
  int vnw_ctr = v2;    /* Real-time variable for nowait loop */

  init_mri();

  nloops=100; duration=1; level=10.0;

  obl_shapedgradient("mygrad",duration,0,level,0,NOWAIT);

  nwloop(nloops,vnw,vnw_ctr);  /* nwloop does initval(nloops,vnw); */

    /* Pulse sequence statements lasting 10 ms go here */

  endnwloop(vnw_ctr);          /* End nowait loop */
}
```

Implications of seqcon settings

The standard prescription of loops in a sequence is

```
#include "sgl.c"

pulsesequence()
{
  int vne = v1, vne_ctr = v2, vns  = v3, vns_ctr  = v4;
  int vnv = v5, vnv_ctr = v6, vnv2 = v7, vnv2_ctr = v8;

  init_mri();

  peloop2(seqcon[3],nv2,vnv2,vnv2_ctr);   /* Start 2nd phase encode loop */
    peloop(seqcon[2],nv,vnv,vnv_ctr);     /* Start phase encode loop */
      msloop(seqcon[1],ns,vns,vns_ctr);   /* Start multi-slice loop */
        initval(ne,vne);                  /* Initialize vne */
        loop(vne,vne_ctr);                /* Start multi-echo loop */

          /* acquisition of echo */

        endloop(vne_ctr);                 /* End multi-echo loop */
      endmsloop(seqcon[1],vns_ctr);       /* End multi-slice loop */
    endpeloop(seqcon[2],vnv_ctr);         /* End phase encode loop */
  endpeloop(seqcon[3],vnv2_ctr);          /* End 2nd phase encode loop */
}
```

seqcon = 'ccccn'

The looping order is the order in the sequence, then array elements:

echo, multi-echo, multi-slice, phase encode, 2nd phase encode, array elements

same data block

different data blocks

seqcon = 'csccn'

The looping order depends on the parameter order in the `array` string:

echo, multi-echo, phase encode, 2nd phase encode, multi-slice, array elements

same data block

different data blocks

echo, multi-echo, phase encode, 2nd phase encode, array elements, multi-slice

same data block

different data blocks

seqcon = 'cscsn'

The looping order depends on the parameter order in the `array` string:

echo, multi-echo, phase encode, multi-slice, array elements, 2^{nd} phase encode

same data block — different data blocks

echo, multi-echo, phase encode, array elements, multi-slice, 2^{nd} phase encode

same data block — different data blocks

seqcon = 'csssn'

The looping order depends on the parameter order in the `array` string:

echo, multi-echo, multi-slice, array elements, phase encode, 2^{nd} phase encode

same data block — different data blocks

echo, multi-echo, array elements, multi-slice, phase encode, 2^{nd} phase encode

same data block — different data blocks

seqcon = 'ccssn'

echo, multi-echo, multi-slice, array elements, phase encode, 2^{nd} phase encode

same data block — different data blocks

seqcon = 'ncsnn'

echo, multi-slice, array elements, phase encode — Standard 2D multislice

same data block — different data blocks

seqcon = 'nscnn'

echo, phase encode, array elements, multi-slice — SSFP type 2D multislice

same data block — different data blocks

Segmentation, nseg and etl

Segmentation is typically performed with the looping structure in the Fast Spin Echo (`fsems`) The parameter `etl` specifies the echo train length.

```
#include "sgl.c"

pulsesequence()
{
  double nseg;

  int vseg     = v1, vms_slices = v2, vetl     = v3;
  int vseg_ctr = v4, vms_ctr    = v5, vetl_ctr = v6;

  init_mri();      /* reads the echo train length etl */

  nseg = nv/etl;   /* Figure the number of segments */
  initval(nseg,vseg);
  peloop(seqcon[2],nseg,vseg,vseg_ctr);    /* Start phase encode loop */
    msloop(seqcon[1],ns,vms_slices,vms_ctr);   /* Start multi-slice loop */
      initval(etl,vetl);                       /* Initialize vetl */
      loop(vetl,vetl_ctr);                     /* Start multi-echo loop */

        /* acquisition of echo */

      endloop(vetl_ctr);                       /* End multi-echo loop */
    endmsloop(seqcon[1],vms_ctr);              /* End multi-slice loop */
  endpeloop(seqcon[2],vpe_ctr);            /* End phase encode loop */
}
```

The `setloop` macro sets `nf` for one pass through the segmented loop for a profile.
Parameters `nseg` (number of segments) and `etl` (echo train length) are used.
`nseg` takes precedence, if it exists, such that `etl = nv/nseg`.

No assumption is made in the `setloop` macro for how `nf` should be set for a full image.
That is left to the `'prep'` macro of the specific sequence.

Data Transfer to Host and nfmod

The default mode of data transfer is for the console to acquire each block of data and, once acquisition of the data block is complete, transfer it across to the host.

There are two modes of acquisition that can be selected by the parameter `dp`
`dp='y'` Each data point is 32-bit float (4 bytes). This is the default
`dp='n'` Each data point is 16-bit int (2 bytes).

Each DDR has 64 Mb of memory for data.

Compressed loops can easily fill the DDR memory, e.g. a basic

256 × 256 matrix, 30 slices, 6 echoes, `dp='y'` $\longrightarrow$ $2 \times 256 \times 256 \times 30 \times 6 \times 4 = 94.4$ Mb

The factor of 2 is for a pair of data points (real and imaginary)
The factor of 4 is for each data point being 4 bytes

Methods for alleviating the problem are:

1. Set a parameter `nfmod` to dictate how often traces of data should be transferred from console to host.
2. Set a loop to be a standard loop.
3. Set `dp='n'` (only saves a factor of 2).

nfmod

if parameter `nfmod` does not exist within a parameter set it can be created on the command line with

```
create('nfmod','integer')
```

`nfmod` sets the number of traces that should be acquired before data is transferred to the host after which the DDR data memory can be reused.

`nfmod` must be a factor of the total number of traces in a block.

`nfmod=1` has been tested fairly extensively with a good degree of success.
Using `nfmod=1` requires no logic to ensure it is a factor the total number of traces in a block.

The major limitation in using `nfmod` is that it does not work if multiple transients are averaged, which is often exactly what is required to achieve adequate SNR for large data sets.

Averaging, il and bs

The parameter `nt` sets the number of transients to be acquired – i.e. the number of repetitions or scans in the experiment.

By default each data block is averaged in the DDR of the console and then transferred to the host before acquisition of the next data block starts.

The parameter `bs` can be used to periodically (every `bs` transients) transfer blocks of data from the console to the host during averaging.

By setting `bs='n'`, block size storage is disabled, and data are stored on the host only when averaging of the block is complete. If an acquisition is aborted prior to termination, the data will be lost.

The interleave flag `il` controls experimental interleaving in arrayed experiments.
The default is for interleaving to be disabled (`il='n'`).

When interleaving is active (`il='y'`), `bs` transients are acquired for each member of the array, followed by `bs` more transients for each member of the array, and so on, until `nt` transients have been collected for each member of the array. As such, `il` is only relevant if `bs` is less than `nt`.

Multiple Receivers

When data is acquired from multiple receivers the data from each receiver goes into a separate data block in the `fid` file. For acquisition of data using four receivers the `fid` file is ordered as follows

File Header	DDR 1 1st block	DDR 2 1st block	DDR 3 1st block	DDR 4 1st block	DDR 1 2nd block	DDR 2 2nd block	DDR 3 2nd block	etc

Current Limitations

1. `nfmod` does not work for `nt > 1`.
   `nfmod` is useful for large data sets that are too big for the DDR memory.
   Large data sets often require `nt > 1` to achieve adequate SNR.

2. `nwloop` functions can not be nested.

3. There is not a good mechanism to set the precedence of standard phase encode loops amongst other array elements.