NMR pulse sequence diagrams, analogy and symbolic calculation: Implementation

Michael P. Barnett,[*]
Meadow Lakes, Hightstown, NJ 08520,

February 28, 2009

Introduction

This report assumes a general familiarity with the working paper "NMR pulse sequence diagrams, analogy and symbolic calculation". The two statements that follow show the simplest way to use APSEQ.

```
pulseSeq[trivseq] = {ch[H1],del[t],p[180]};(* define a pulse sequence *)

drawPulseSequence[trivseq](* construct and write trivseq.ps *)
```

An `includegraphics` statement imports the file:

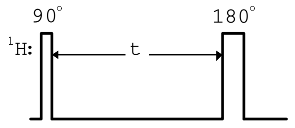

Figure S1.

Figure labels in this supplement begin with the letter "S". Figure labels that are entirely numeric, *e.g.* Figure 1.1, refer to the working paper. This supplement provides information for users who are interested in how APSEQ has been coded, in particular, to make modifications and extensions. The successive sections of this supplement describe the action of the `drawPulseSequence` procedure and its supporting procedures by tracing the action for

1. the very simple pulse sequence in Figure S1,
2. the further needs of the refocused HETCOR pulse sequence of Figure 1.1,
3. gradient pulses, by reference to the INEPT pulse sequence of Figure 1.2,
4. spin locks, by reference to the basic 2D TOCSY sequence of Figure 1.3.

[*]michaelb@princeton.edu
[†]MATHEMATICA is a registered trademark of Wolfram Research Inc.

1 Processing the pulse sequence in Figure S1

1.1 Overview of drawPulseSequence

The `drawPulseSequence` procedure

1. under comment 1, carries out some preliminaries, that include
   (a) checking for a valid name,
   (b) assigning `pulseqRep` to the pulse sequence specified above,
   (c) invoking `writeNotes` that can be ignored for the present,
   (d) checking that the sequence begins with a channel specification,
   (e) setting `channels` to the list of channels, *i.e.* `{H1}`,
   (f) setting `channelCount` to the number of channels, *i.e.* 1,
   (g) setting `inxGz` to 0 because there is not a gradient pulse channel,
   (h) unsetting `currentAxis`, which specifies the subscript of a pulse angle.
2. under comment 1.1, sets `indixes` to `{1}` for reasons explained later,
3. under comments 2–12 (most of the procedure), constructs in parallel
   (a) the list `latent[i]`, `i`= 1, . . . ,`channelCount`, that specifies the outlines of the pulse sequences and other events,
   (b) the list `latentX[i]`, `i`= 1, . . . ,`channelCount` that shows the duration of each event by an appropriate symbol centered between arrows,
4. under comment 13, uses the procedure `gml` to convert these `latent` and `latentX` lists to a single list of graphics objects called `activeAll`, by applying the procedure ,
5. under comment 14, passes `activeAll` to DISPLAY, which uses the MATHEMATICA `Display` function to construct the PostScript representation and to write this as output.

1.2 The latent and latentX lists

The detailed process that constructs these lists was designed for multi-channel pulse sequences, and a lot of its detail is unnecessary for a one-channel sequence. The explanation of the process is deferred, accordingly, to §2 *et seq.* For the example in hand, the process constructs

```
latent[1] => {$delay[5], $pulse[$pd$[90, currentAxis]], $delay[80],
              $pulse[$pd$[180, currentAxis]], $delay[20]}

latentX[1] => {raise[30], $atomLabel[H1], space[5], space[5],
              alignAndFill[betweenArrows, {t}, 80], space[10]}
```

These are read, in conjunction, as follows.

1. `raise[30]` in `latentX[1]` raises the drawing cursor 30 points above the baseline of the diagram, for the labels that specify channels and times.
2. `$atomLabel[H1]` produces the label ^{1}H: that identifies the channel.
3. `$delay[5]` in `latent[1]` gives a 5 point segment along the baseline..
4. `space[5]` in `latentX[1]` spaces the drawing cursor by 5 points to stay vertically aligned.

5. `$pulse[$pd$[90, currentAxis]` in `latent[1]` provides the rectangular depiction of a 90° pulse, suitably labeled. Because `currentAxis` is unassigned when this is interpreted, there is no x, y or z subscript.
6. The second `space[5]` in `latentX[1]` moves the drawing cursor by another 5 points, to stay vertically aligned.
7. `$delay[80` in `latent[1]` continues the outline with an 80 point segment along the baseline.
8. `alignAndFill[...]` in `latentX[1]` centers `t` between arrows that span the 80 point segment of the baseline.
9. The second `$pulse` item in `latent[1]` draws the 180° pulse.
10. `space[10]` in `latentX[1]` moves the drawing cursor by another 20 points, to stay vertically aligned.
11. `$delay[20]` item in `latent[1]` draws the final segment of the baseline.

1.3 The activeAll list

The `latent` and `latentX` lists are converted to MATHEMATICA graphics primitives in the section of `drawPulseSequence` that follows comment 13. Here, `channelCount` is 1, and `indixes[[1]]` is 1. An explanation of the role of indixes, in controlling the relative vertical arrangement of the channels when there are more than 1 is explained later. The index `k` takes the single value 1. The second argument of the`gml` expressions reduces to `{30, 580}`. This initializes the coordinates (`xmLocal`, `ymLocal`) that position the graphics to (30, 580). The `gml` expressions interpret the successive items in the preceding lists as follows, when `scale` is 1.

1. `raise[30]` increases `ymLocal` from 580 to 610.
2. `$atomLabel[H1]` is converted to the 3 graphics objects that provide the superscript 1, the letter H, and the colon, respectively.
```
{Text[FontForm[1, {Courier, 6.}], {19.4, 614.8}, {-1, -1}],
  Text[FontForm[H, {Courier, 10.}], {23., 610}, {-1, -1}],
  Text[FontForm[:, {Courier, 10.}], {29., 610}, {-1, -1}]}
```
3. `$delay[5]` gives `Line[{{30, 580}, {35, 580}}]`.
4. `space[5]` increases `xmLocal` from 30 to 35.
5. `$pulse[$pd$[90, currentAxis]` gives the graphics objects that
   (a) draw the outline of the pulse,
   (b) provide the numerical value of the angle, *i.e.* 90,
   (c) draw the degree symbol.

   These are displayed on the next four lines
```
{Line[{{35, 580}, {35, 620}, {40, 620}, {40, 580}}],
 {Text[FontForm[90, {Courier, 10.}], {31.5, 625}, {-1, -1}],
   AbsoluteThickness[0.1], Circle[{43.5, 631.667}, 1],
   AbsoluteThickness[1]}}
```
6. The second `space[5]` increases `xmLocal` from 35 to 40.
7. `$delay[80]` gives `Line[{{40, 580}, {120, 580}}]`.
8. `alignAndFill[betweenArrows, {t}, 80]` gives the graphics objects that draw, respectively,
   (a) the upper part of the left arrow head,
   (b) the left arrow shaft,

(c) the lower part of the left arrow head,
(d) the symbol for the delay,
(e) the upper part of the right arrow head,
(f) the left arrow shaft,
(g) the lower part of the right arrow head.

These are displayed on the next 7 lines

```
{Polygon[{{40, 610}, {43, 612}, {43, 610.3},
           {74.5, 610.3}, {74.5, 609.7},
           {43, 609.7}, {43, 608}, {40, 610}}],
 {Text[FontForm[t, {Courier, 10.}], {77., 610}, {-1, -1}]},
  Polygon[{{120, 610}, {117, 612}, {117, 610.3},
             {85.5, 610.3}, {85.5, 609.7},
             {117, 609.7}, {117, 607}, {120, 610}}]]}
```

9. `$pulse[$pd$[180, currentAxis]]` produces another expression similar to item 5, but with the horizontal line segment twice as long, and the number 180 instead of 90.
10. `$delay[20]` gives another `Line` that extends the baseline.

1.4 Writing the PostScript file

The `display` statement (see above) for the example in hand is

```
display[trivseq, activeAll]
=>
Display["!psfix > trivseq.ps", Show[Graphics[
  {AbsolutePointSize[abp], AbsoluteThickness[abt], activeAll} // Flatten,
  PlotRange -> {{0, 6.5 * 72}, {0, 9 * 72}}, AspectRatio -> 9/6.5 ]]]
```

The `PlotRange` and `AspectRatio` make the distance of 1 unit in the specification of a diagram correspond to 1 point on the printed page, when the file is incorporated in a LaTeX document by an `includegraphics` statement with `scale=1`.

1.5 The gml function

In the invoking expression `gml[items, {xstart, ystart}]`

1. `items` is a `latent` or `latentX` list in a production run,
2. `{xstart, ystart}` position the pulse sequence diagram.

The `gml` function constructs `moutList` by

1. initializing this to an empty list, then
2. for each element of `items`, appending graphic objects if appropriate and adjusting `xmLocal` and `ymLocal`, the dynamic values of the drawing coordinates.

The actions for the successive elements of `items` are performed by

1. a `Do` loop indexed by `iii`, in which the item under current attention is assigned to `currentItem`,
2. a `Switch` on the head of `currentItem`.

For the example in hand, this switch leads, in turn, to the value (of the head)

1. `raise`, which has the direct effect of incrementing `ymLocal`.
2. `$atomLabel`, which appends, to `moutList`, the value of `atomLabel[H1]`, that the function `atomLabel` constructs. This use of the `$` symbol concatenated to a function name f, to latentize f until (`xmLocal`, `ymLocal`) have appropriate values, is a core item of the design of APSEQ. The internal action of `atomLabel` is explained later.
3. `$delay`, which appends, to `moutList`, the value of `delay[5]` that the function `delay` constructs.
4. `space`, which has the direct effect of incrementing `xmLocal`.
5. `$pulse`, which appends the value of `pulse[$pd$[90, currentAxis]` that the function `pulse` constructs, to `moutList`, .
6. another `space`, that acts as described for the first `space`,
7. another `$delay`, that acts as described for the first `$delay`.
8. `alignAndFill`, that appends to `moutList`, the graphics objects that construct a label between arrows that span the specified width. The steps are explained at the end of the next subsection.

1.6 Functions for diagram modules

For the current example:

atomLabel: this introduces the `gtl[items, {xstart, ystart}]` function that converts the list `items` that specifies a piece of text into a list of `Text` expressions. The `gtl` function returns this list. Also, it assigns the width of the text to `totalWidth`. The object `scorpic[prep]` is the width of the short baseline segment that begins the depiction of a pulse sequence, on the left of the exciting pulse. The rationale of this expression is explained in the account of `gtl` below. The rest of `atomLabel` is self-explanatory.

delay: this wraps `extend` which is self-explanatory.

pulse: the arguments are wrapped in the head `$pd$`, that is redundant but harmless, for reasons that were dropped but may be re-introduced. The visual width of a 90° pulse, in printers points, and its height, are denoted by `pwu` and `pulseHeight`. These have default values that can be overridden. The `tentativeLabel` is constructed to determine the width of the label, so that it can be centered over the pulse.

alignAndFill: this uses `gtl` to convert the symbol for the duration of the event into graphics primitives, and to determine its width. The action, that also includes the use of `centerBetweenArrows` and the subsidiary `leftArrow` and `rightArrow`, is self-explanatory.

1.7 Conversion of text to graphics

In APSEC, the procedure `gtl` constructs the graphic objects that represent text. The most elementary action of `gtl` is shown by

```
gtl[{"f", "g"}, {100, 500}]
=>
{Text[FontForm["f", {"Courier", 10.}], {100, 500}, {-1, -1}],
  Text[FontForm["g", {"Courier", 10.}], {106., 500}, {-1, -1/2}]}
```

These statements concatenate the letters `f`and `g` starting at $x = 100$, with their baselines on $y = 500$. This is shown by

```
display["x02", {fg, Line[{{0, 500}, {30, 500}}]}, (* abt *) 0.1  ]
```

where `abt` is the absolute line width. This produces

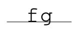

Figure S2.

The offset `{-1, -1}` for the letter `f` is in obvious accord with the description of this paramater in standard texts on MATHEMATICA. But using this offset for `g`, `j`, or other letters with descenders puts the tip of the descender on $y = 500$. Hence the offset `{-1, -1/2}` for the `g`. The placement of letters by `gtl` is supported by the functions

1. `width[`*letter*`, {`*fontName*`, `*fontSize*`}]`, that gives the width of the bounding box of the letter, and
2. `voffset[`*letter*, *fontName*`]`, that gives the y offset, that is -1 for most letters and $-\frac{1}{2}$ for letters with descenders.

The default font is Courier 10. Greek symbols are in Mathematica1.

The representation of

1. each symbol, such as t, t_1, τ_m, and
2. each expression, such as $1/J$, $2/J$, $1/(2J)$,

that specifies the time duration of an event, is assigned to `txtForm[`*s*`]` for interpretation by `gtl`. Thus

```
txtForm[taum] =
 {{"Mathematica1", 10}, "t", {"Courier", 10}, subscript, m, basescript};
```

Then

```
mixedFonts =
 gtl[ {"decorated Greek text: ", txtForm[taum]}//flatten[1], {10, 500}];
graphics["x04"] = display["x04", mixedFonts]
```

generates the PostScript file `x04.ps`that produces the following line *via* an `\includegraphics` statement.

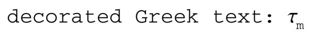

Figure S3.

The value of `gtl[textList, {x, y}]` is the list of graphic objects that represents the concatenation of the items in `textList`, with the lower left hand corner at `{x, y}`. Each item is either

1. a string,
2. a pair `{`*fontName*, *fontSize*`}`,
3. `subscript` or `superscript` or `basescript`,
4. `degree` for the symbol $\circ$.

In particular, `t1`, `t2`, `t3` are converted to lists that represent t_1, t_2, t_3.

The variable `currentFontSize` specifies font size in every `Text` statement that is used in `gtl`. It is

1. initialized to `fontScale * $DefaultFont[[2]]` under comment 1, and
2. changed by an item in the input list, by a statement under comment 5 that scales the input specification correspondingly.

The variable `fontScale` is initialized to 1.

The procedure `gtl` returns the appropriate list of graphic objects. Also, the total width of their concatenation is accessible to the invoking procedures as `totalWidth`. The coding is direct and mnemonic.

1.8 Centering between arrows, continued

1. Each symbol u that denotes the duration of an event is assigned a physical width for its depiction in the graphics. This is denoted by `forpic[`u`]`.
2. The list of symbols that are assigned `forpic` values by the user is constructed, in reverse alphabetic order, by the utility `domain`. At run time, it is assigned the name `sdList`.
3. If the members of `sdList` are $v_1, v_2, \ldots$, and s is an arbitrary string, then `inpic[`s`]` returns the result of replacing each occurrence of the v_i in s by their `forpic` values.
4. `leftArrow` and `rightArrow` construct left and right pointing arrows with tips at the position specified by the 1^{st} and 2^{nd} arguments and length specified by the 3^{rd}.
5. `centerBetweenArrows[xl, xr, y][text]` creates the list of graphic objects that represent `text` centered between `xl` and `xr` at height `y`.

The `forpic` values are actually wrapped in the `scorpic` function that simply multiplies them by `scale`. This is initialized to 1, but can be reassigned to change the size of an output diagram. The fonts are scaled by `fontScale`, which is initialized to 1. The coding is direct and mnemonic. It uses `gtl` to convert text to a list of graphic objects.

1.9 Scaling

By default, ASPEQ makes the distance of 1 unit in the specification of a diagram correspond to 1 point on the printed page. Actually, it corresponds to `scale` points, where `scale` defaults to 1. The factor is applied

1. by wrapping `forpic` in `scorpic`,
2. by scaling the dimensions of the arrow tips in `leftArrow` and `rightArrow` (the arrow lengths are scaled before these procedures are invoked),
3. in the assignments of `pulseHeight` and the other dimensions of pulses, following the comment `pieces of pulse sequence design`,
4. in the positioning of the degree symbol in `pulse`,
5. in the statements that compute positions and dimensions in `grad`,
6. in the positioning of the depictions of the channels, in the `gml` expressions following comment 13 in `displayPulseSequence`.

The presence of the scale factor will be taken for granted throughout the rest of this explanation.

The symbols that specify the duration of delays and other events are set in 10 point font by default. This is scaled by `fontScale` that defaults to 1. The subscript and superscript font sizes are scaled by this factor, too.

2 Processing the HETCOR pulse sequence

2.1 Preliminaries

The refocused HETCOR pulse sequence is defined and drawn, in §2 of the working paper by

```
pulseSeq[hetcorRefocused] =
 {ch[H1, C13], del[t1/2], {, p[180]}, del[t1/2], del[1/(2J)] ,
    p[90], del[1/(3J)], {dec, fid}[taq]};
```

```
drawPulseSequence[hetcorRefocused]
```

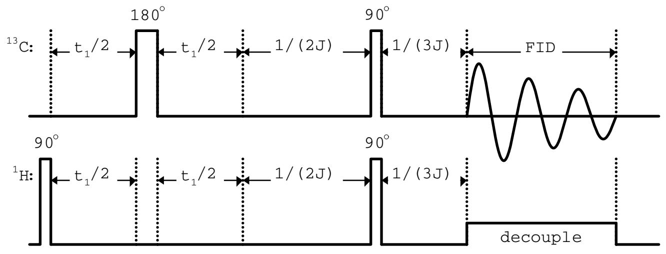

Figure S4 (copy of Figure 2.1). Refocused HETCOR.

This introduces a major feature of the APSEQ coding. In the `pulseSeq` representation, parallelism is represented either

1. by braces, as in `{dec, fid}[taq]`, or
2. when the same action is applied on all channels simultaneously, by the absence of braces, as in the `del` items and `p[90]` in the current example.

The procedure `drawPulseSequence` converts this representation of parallelism to the separate `latent` and `latentX` lists for each channel. A large part of `drawPulseSequence` works with the brace representation,

1. converting items that are common to all channels from the unbraced form such as `p[90]` to `{p[90], p[90]}`, and
2. and changing null items, within braces, to items that lead *e.g.* to the dotted pulse shadows.

2.2 Action under comments 1–3

These tactics are described more fully in the detailed description of the execution of the `drawPulseSequence` procedure, for refocused HETCOR, that follows.

1. The statements under comment 1 assign:
   (a) `pulseqRep` to the representation of the pulse sequence shown above,
   (b) `channels` to `{H1, C13}`,
   (c) `channelCount` to 2,
   (d) `inxGz` to 0.

2. The statements under comment 1.1 assign `indixes` to `{2, 1}`, to position the ^{1}H channel below the ^{13}C channel, (in general, `indixes[`*k*`]` is the position of the channel that is in *k*th place in the `ch` expression, counting downwards from the top of the display),
3. The statements under comment 2 assign
   (a) `latent[1]` to
   `{$delay[scorpic[prep]] , $pulse[$pd$[90, currentAxis]]}`,
   (b) `latent[2]` to `{$delay[5], $pulseShadow[$pd$[90]]}`,
   (c) `latentX[1]` to
   `{{raise[30], $atomLabel[H1], space[5], space[5]}}`,
   (d) `latentX[2]` to
   `{{raise[30],$atomLabel[C13],space[5],space[5]},$sdots[]}`.

   The only novel item is the `$sdots[]`, that provides the vertical line of dots in the ^{13}C channel aligned with the right edge of the excitation pulse.
4. The statements under comment 3
   (a) assign `priorPulseqItem` to `{Null, Null}[0]` for the comparison that puts a vertical line of dots between consecutive delays, and then
   (b) work through the 2nd through 7th elements of the pulse sequence representation, as explained in the following subsections.

2.3 The Do loop processing of del[t1/2]

When `inx` is 2 in the `Do` loop under comment 3, `pulseqItem` is `del[t1/2]`, designating a delay of `t1/2` on both channels. The statements under comments 4–10x act as follows.

(a) Under comment 4, `pulseqItem` is changed to `{del, del}[t1/2]`.
(b) Under comments 5–9 there is no effect.
(c) Under comment 10,
   i. `pulseqItem` matches `{__}[_]` because of the action under comment 4, so the error exit is bypassed,
   ii. In the `Which`, the criterion for the first case is not satisfied, because `pulseqItem` does not contain any code associated with a pulse. The criterion for the second case is satisfied, for the same reason. Consequently, `currentDuration` is set to `t1/2`.
   iii. The final statement has no effect (it seems part of an obsolete handling of the `insertGradientPulse` command, but has not been deleted yet, as a matter of caution).
(d) Under comment 11: the `Do` loop is executed with `ii` successively 1 and 2. Each time, `currentInnerItem` is assigned to `del`.
(e) Within the `Switch`, the case `del` acts as follows.
   i. statement `(*1*)` appends `$delay[40]` to both `latent` lists,
   ii. statement `(*2*)` has no effect because `currentDuration` is `t1/2`.
   iii. The condition in statement (*3*) is satisfied, so the inner statements (*4*) and (*5*) are evaluated.
   iv. The condition in statement (*4*) is not satisfied (see action under comment 3 above), so this has no effect.

v. Statement (*5*) appends, to `latentX[1]` and `latentX[2]`,
`alignAndFill[betweenArrows, {t,subscript,1,basescript,/2}, 40]`.

(f) After the `Switch`, `priorPulseqItem` is assigned to `{del,del}[t1/2]`.

2.4 The Do loop processing of {Null, p[180]}

When `inx` is 3 in the `Do` loop under comment 3, `pulseqItem` is `{, p[180]}`, designating a 180° pulse on channel 2. The statements under comments 4 to 10x act as follows.

(a) Under comments 4 and 5, no action.
(b) Under comment 6 the `MatchQ` criterion is met, so
  i. `pulseItem` is set to `p[180]`,
  ii. `pulseAngle` is set to `180`,
  iii. because the length of `pulseItem` is 1, `currentAxis` is left unassigned,
  iv. `pulseqItem` is changed to `{pulseShadow, p}[180]`,
  v. because the pulse sequence does not include a gradient pulse channel, `inxGz` is 0 and the conditional statement has no effect.
(c) Under comments 7 and 8, the criteria for action are not met..
(d) Under comment 9, there are no `Null` elements in `pulseqItem`, so it is not altered.
(e) Under comment 10, the criterion for the first case in the `Which` is met, so `pulseAngle` is set to `180` (redundant but harmless), and `currentDuration` is unset.
(f) The final statement has no effect (see previous subsection).

5. In the `Do` loop under comment 11, `currentInnerItem` is `pulseShadow` when `ii` is 1. The `Switch`, accordingly, appends

   (a) `{$udots[], $pulseShadow[$pd$[180]], $udots[]}` to `latent[1]`,
   (b) `space[10]` to `latentX[1]`.

   Then, when `ii` is 2, `currentInnerItem` is `p`, and The `Switch` appends

   (a) `$pulse[$pd$[180, currentAxis]` to `latent[2]`,
   (b) `space[10]` to `latentX[2]`.

6. After the `Switch`, `priorPulseqItem` is assigned to `{pulseShadow,p}[180]`.

2.5 The Do loop processing of the next two delays

When `inx` is 4 and 5 in the `Do` loop under comment 3 then, respectively,

1. the second `del[t1/2]` is processed in the same way as the first, except that at the end, `priorPulseqItem` is assigned to `{del, del}[t1/2]`,
2. the first `del[1/(2J)]` is processed in the same way as the first delay, except that

   (a) in statement (*4*) in the case `del` in the `Switch`, the criterion is satisfied, and `sdots[]` is appended to `latentX[1]` and to `latentX[2]`, to provide a vertical line of dots between the two consecutive delays,
   (b) at the end, `priorPulseqItem` is assigned to `{del, del}[1/(2J)]`.

2.6 The Do loop processing of p[90]

When `inx` is 6 in the `Do` loop under comment 3, `pulseqItem` is `p[90]`, designating a 90° pulse on both channels. The statements under comments 4–10x act as follows.

1. Under comment 4, no effect.
2. Under comment 5,
   (a) the criterion in the `MatchQ` is satisfied,
   (b) `pulseAngle` is set to 90,
   (c) `currentAxis` is left unassigned,
   (d) `pulseqItem` is changed to `{p, p}[90]`,
   (e) the condition for presence of a gradient pulse channel is not met, so the final statement has no effect.
3. Under comment 6, no effect, because neither `p` has a direct argument.
4. Under comments 7, 8, 9, no effect, because `If` conditions are not met.
5. Under comment 10, the criterion for the first case in the `Which` is met, so `pulseAngle` is set to `90` (redundant but harmless), and `currentDuration` is unset.
6. The final statement under comment 10 has no effect (see §2.2).
7. In the `Do` loop under comment 11, `currentInnerItem` is `p` when `ii` is 1 and again when it is 2. The `Switch`, accordingly, appends
   (a) `$pulse[$pd$[90, currentAxis]` to `latent[1]` and to `latent[2]`,
   (b) `space[10]` to `latentX[1]` and to `latentX[2]`.
8. Following the `Switch`, `priorPulseqItem` is assigned to `{p, p}[180]`.

2.7 The Do loop processing of del[1/(3J)]

When `inx` is 7 in the `Do` loop under comment 3, `pulseqItem` is `del[1/(3J)]`. This is processed in just the same way as the 1st and 2nd delays.

2.8 The Do loop processing of dec, fid[taq]

When `inx` is 8 in the `Do` loop under comment 3, `pulseqItem` is `{dec, fid}[taq]`, designating decoupling on channel 1 and acquisition on channel 2. The statements under comments 4–10x act as follows.

1. Under comments 4–9, no action, because the criteria in the respective conditional statemenst are not met.
2. Under comment 10, the criterion in the first case of the `Which` statement is not met. The criterion in the second case is met. Consequently, `currentDuration` is set to `taq` and `pulseAngle` is unset. The assignment of `currentDuration` at this juncture is not redundant here, even though it was redundant but harmless for earlier items.
3. In the `Do` loop under comment 11, when `ii` is 1, the case `dec` appends `{$udots[], $decouple[70], $udots[]}` to `latent[1]`, and `space[70]` to `latentX[1]`, because the default value of `forpic[taq]` is 70.
4. When `ii` is 2, the case `fid` appends `{$udots[],$fid[70, 3],$udots[]}` to `latent[2]` and `alignAndFill[betweenArrows,{FID},70]` to the label list `latentX[2]`.

2.9 Finishing the process

The `{dec, fid}[taq]` item ends `pulseqRep`. Action now falls to the statements under comment 12–14.

1. The statement under comment 12 appends `$delay[20]` to `latent[1]` and `latent[2]`.
2. The statements under comment 13 construct `activeAll` by applying `gml` to the `latent` and `latentX` lists. This is discussed in the next subsection.
3. The statement under comment 14 writes the POSTSCRIPT file for the refocused HETCOR diagram.

2.10 The latent and latentX lists

The complete `latent[1]` that is formed in the process that has just been described, is

```
{$delay[5], $pulse[$pd$[90, currentAxis]], $delay[40], $udots[],
 $pulseShadow[$pd$[180]], $udots[], $delay[40], $delay[100],
 $pulse[$pd$[90, currentAxis]], $delay[200/3], $udots[],
 $decouple[70], $udots[], $delay[20]}
```

This contains three novel elements.

1. `$udots[]` is converted by `gml` to the function name `udots`, that is appended to `moutList` and evaluated automatically, *in situ*, to give a list of `Point` graphics objects that draw a dotted vertical line, with the same height as the depiction of a pulse. For the example in hand, this is `{Point[{80, 520}], Point[{80, 522}],...Point[{80, 560}]}`.
2. `$pulseShadow[$pd$[180]` is converted by `gml` to `pulseShadow[180]`, that is evaluated automatically *via* `extend[10]` to provide the `Line` that extends the baseline by the width of the pulse. The three items `$udots[]`, `$pulseShadow[$pd$[180]]`, and `$udots[]` thus shadow the pulse by vertical dotted lines that are aligned with the vertical edges of the pulse, joined along the baseline.
3. `$decouple[70]` is processed similarly, appending `decouple[70]`, that is evaluated automatically, to `moutList`. This uses `decouple[time]` to construct the list of graphic objects that depicts decoupling. This procedure is self-explanatory. It produces a `Line` expression for the outline, and a `Text` expression for the word “decouple”.

The list `latentX[1]` for the current example is

```
{raise[30], $atomLabel[H1], space[5], space[5],
  alignAndFill[betweenArrows, {t, subscript, 1, basescript, /2}, 40],
  space[10],
  alignAndFill[betweenArrows, {t, subscript, 1, basescript, /2}, 40],
  $sdots[], alignAndFill[betweenArrows, {1/(2J)}, 100], space[5],
  alignAndFill[betweenArrows, {1/(3J)}, 200/3], space[70]}
```

The only novel element is `$udots[]`. This has the same effect as `$sdots[]`, acting *via* the function `sdots`. Both are needed, because `udots` draws upward from the baseline of the depiction of a channel, and `sdots` draws downward from the level of the labels that are centered between arrows.

The list `latent[2]` for the current example is

```
{$delay[5], $pulseShadow[$pd$[90]], $delay[40],
  $pulse[$pd$[180, currentAxis]], $delay[40], $delay[100],
  $pulse[$pd$[90, currentAxis]], $delay[200/3], $udots[],
  $fid[70, 3], $udots[], $delay[20]}
```

This contains two novel elements.

1. `$pulseShadow[$pd$[90]]`is converted by `gml pulseShadow[$pd$[90]]`, that is appended to `moutList` and evaluated *in situ*, *via* `extend`, to the `Line` expression that draws the line segment with the same width as the pulse that is being shadowed.
2. `$fid[70, 3]` is converted correspondingly to `fid[70, 3]` and the value of this is appended to `moutList`. The procedure `fid[width, cycles]` constructs the graphic objects that draw an attenuated sine wave, with maximum amplitude 2/3 the pulse height, and the specified number of cycles.

The list `latentX[2]` for the current example is

```
{raise[30], $atomLabel[C13], space[5], space[5], $sdots[], $sdots[],
  alignAndFill[betweenArrows, {t, subscript, 1, basescript, /2}, 40],
  space[10], $sdots[],
  alignAndFill[betweenArrows, {t, subscript, 1, basescript, /2}, 40],
  $sdots[], alignAndFill[betweenArrows, {1/(2J)}, 100],
  space[5], $sdots[],
  alignAndFill[betweenArrows, {1/(3J)}, 200/3],
  alignAndFill[betweenArrows, {FID}, 70]}
```

This contains no novel features.

3 Dealing with gradient pulses

3.1 Preliminaries

The gradient enhanced INEPT pulse sequence is defined and drawn, in §2 of the working paper, by

```
pulseSeq[gradientEnhancedInept] =
 {ch[H1, C13, Gz], del[1/(4J)], p[180],
   del[1/(4J)], {p[90],}, gp[], {, p[90]}, {, fid}[taq]} ;

drawPulseSequence[gradientEnhancedInept]
```

Hence

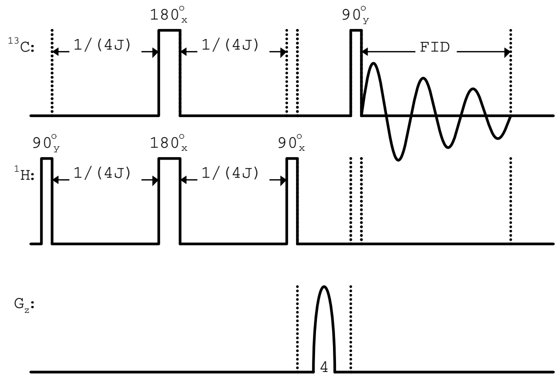

Figure S5 (copy of Figure 2.2). Gradient enhanced INEPT.

This example introduces the pieces of APSEQ code that handle gradient pulses. These are explained by describing the treatment of the INEPT pulse sequence by `drawPulseSequence`. Obvious repetitions of actions that have been described already are taken for granted.

3.2 Initialization

The statements under comments 1, 1.1, 2 and 3

1. set `chanCount` to 3,
2. set `inxGz` to 3,
3. set `indixes` to `{2, 1, 3}`, to display the channels in the order ^{13}C, ^{1}H, G$_z$, going downwards on the page,
4. put `$sdots[]` in `latentX[2]`, to provide the dotted line, that is aligned vertically with the right edge of the exciting pulse, in the 13C channel but not in the ^{1}H or G$_z$ channels,
5. set `priorPulseqItem` to `{Null, Null, Null}[0]`,
6. begin the `Do` loop through the successive elements of `pulseqRep`.

3.3 Processing the first del item

When `inx` is 2, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is `del[1/(4J)]`.
2. The statement under comment 4 converts this to `{del, del, nullShadow}[1/(4J)]`.
3. The statements under comments 5 to 10 have no effect.
4. The `Do` loop under comment 11 is executed.
   (a) When `ii` is 1 and 2, the item `del` is processed as described earlier.
   (b) When `ii` is 3, the the action for the item `nullShadow` appends
      i. `$extend[70]` to `latent[3]`, to extend the base line in the G_z channel, and
      ii. `space[70]`, to space the caption line correspondingly.

   The second action is redundant for present conventions which do not put data in this line for the G$_z$ channel.

3.4 Processing the first p item

When `inx` is 3, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is set to `p[180]`.
2. The statements under comments 4 has no effect.
3. The statements under comment 5
   (a) set `pulseAngle` to 180,
   (b) leave `currentAxis` unassigned,
   (c) convert `pulseqItem` to `{p, p, p}[180]`,
   (d) convert this to `{p, p, nullPulseShadow}[180]`, because `inxGz` is 3.
4. The statements under comments 6 to 9 have no effect.
5. In the statements under comment 10, the first criterion in the `Which` is satisfied, so `currentAngle` is set to 180 and `currentDuration` is unset. The final `If` has no effect.
6. The `Do` loop under comment 11 is executed.
   (a) When `ii` is 1 and 2, action proceeds for the two `p`items, as described earlier.
   (b) When `ii` is 3, the processing of `nullPulseShadow` item appends
      i. `{$pulseShadow[$pd$[180]]}` to `latent[3]`, and
      ii. `space[10]` to `latentX[3]`.

Later, during the formation of `activeAll` by `gml`, `$pulseShadow` is activated to `pulseShadow`. This extends the base line of the G_z channel to keep pace with the pulses. The `space` is redundant but harmless, as mentioned at the end of the preceding subsection.

3.5 Processing the second del

When `inx` is 4, in the `Do` loop that starts under comment 3, `pulseqItem` is `del[1/(4J)]` again. This is processed in the manner described in §3.3 above.

3.6 Processing the {p[90], Null}

When `inx` is 5, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is set to `{p[90], Null}`.
2. The statements under comments 4 and 5 have no effect.
3. The statements under comment 6
   (a) set `pulseItem` to `p[90]`,
   (b) set `pulseAngle` to 90,
   (c) leave `currentAxis` unassigned,
   (d) convert `pulseqItem` to `{p, pulseShadow, nullPulseShadow}[90]`.
4. The statements under comments 7 to 9 have no effect.
5. The statements under comments 10 and 11 have the overall effect of appending
   (a) `$pulse[$pd$[90, currentAxis]` to `latent[1]`,

(b) `{$udots[], $pulseShadow[$pd$[90]], $udots[]` to `latent[2]`,
(c) `{$pulseShadow[$pd$[90]]` to `latent[3]`,
(d) `space[5]` to all three `latentX` lists,

by processes described in earlier subsections. These items lead to the depiction of the pulse in the ^{1}H channel, the vertical dotted lines aligned with its edges in the ^{13}C channel, and the extension of the baseline in the G_z channel.

3.7 Processing a gradient pulse

When `inx` is 6, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is set to `pg[]`. This denotes a gradient pulse of default
2. The statements under comments 4, 5, 6 have no effect.
3. The statements under comment 7
   (a) set `gradientSpec` to `C13` (the default channel for the gradient),
   (b) set `displayStrength` to `False`,
   (c) set `pulseqItem` to `{nullShadow, dottedShadow, gp[C13]}[tGz]`.
4. The statements under comments 8 and 9 have no effect.
5. In the statements under comment 10, the second criterion in the `Which` is met, `currentDuration` is set to `tGz` and `pulseAngle` is unset.
6. In the `Do` loop under comment 11,
   (a) when `ii` is 1 and 2, `currentInnerItem` is `dottedShadow`. This makes the `Switch` append `{$udots[], $extend[70], $udots[]}` to `latent[1]` and `latent[2]`. When these items are processed by `gml`, the graphic objects are appended to `activeAll` to draw vertical dotted lines in the ^{1}H and ^{13}C channels, that are vertically aligned with the edges of the depiction of the gradient pulse. Also, `space[70]` is appended to `latentX[1]` and `latentX[2]`.
   (b) When `ii` is 3, `currentInnerItem` is `gp[C13]`. This makes the `Switch` append `{$udots[],$grad[C13,70],$udots[]}` to `latent[3]`. Nothing is appended to `latentX[3]` because times are not labeled in the gradient channel.

3.8 Processing the {, p[90]}

When `inx` is 7, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is set to `{, p[90]}`.
2. The statements under comment 4 have no effect.
3. The statements under comment 5 change `pulseqItem` to `{pulseShadow, p, nullPulseShadow}[90]`. The final `If` statement is needed, because the subsequent processing requires the `nullPulseShadow`.
4. The statements under comments 6 to 11 follow courses of action that have been described already. These put the items into the `latent` lists that draw the pulse in the ^{13}C channel, shadow it in the ^{1}H channel, and simply extend the baseline to stay aligned in the G_z channel.

3.9 Processing the {, fid}[taq]

When `inx` is 8, in the `Do` loop that starts under comment 3,

1. `pulseqItem` is set to `{, fid}[taq]`.
2. The statements under comments 4 to 8 have no effect.
3. The statements under comment 9 change the `pulseqItem` to `{dottedShadow, fid}[taq]`, to feed the `latent[1]` list later with the keyword that leads to dotted lines in the ^{1}H channel that are aligned with the ends of the acquisition
4. The statements under comment 10 change the `pulseqItem` to `{dottedShadow, fid, nullShadow}[taq]`, to feed `latent[3]` list with keyword to extend the baseline.
5. The statements under comments 10 and 11 follow courses of action that have been described already. These put the items into the `latent` lists that depict acquisition in the ^{13}C channel, extend the baselines in the ^{1}H and G$_z$ channels, and put the dotted lines in the ^{1}H and G$_z$ channel.

3.10 Completing the process

The statements under comments 12, 13 and 14 proceed, in overall terms, as before. The appropriate lists of graphic objects are formed in `activeAll`, and `display` constructs the POSTSCRIPT file. The novel elements in the action of `gml` are the branches to the statements that deal with `nullShadow`, `nullPulseShadow`, `dottedShadow` and `gp[...]`. The effects of the first three have been described already. The `gp` leads to the function`$grad`. This is converted to a `grad` expression, that is evaluated automatically as follows.

3.11 The grad function

This has two forms. The first, `grad[`*atomName*, *duration*`]`, where *atomName* is `C13`, `N15`, . . . , is converted automatically to the form `grad[`*strength*, *duration*`]`, where the strength is 4 for ^{13}C and 10 for ^{15}N, ... By default, the strength is the ratio of the gyromagnetic ratio of the atom that is named to that of a proton. The function `grad[`*strength*, *duration*`]` is the list of graphic objects that draws a semi-ellipse with horizontal axis `xaxisGz` that defaults to 5, and vertical axis `gpHeightScale`×*strength*, with the base line extended uniformly on each side to fill the width *duration*. The switch `displayStrength` determines whether the numerical value of the strength is displayed .

4 Processing a spin lock

4.1 Preliminaries

The 2D TOCSY sequence is defined and drawn, in §2 of the working paper, by

```
pulseSeq[2D, tocsy] = {ch[H1], del[t1] , lock[taum],  fid[taq] }

drawPulseSequence[2D, tocsy]
```

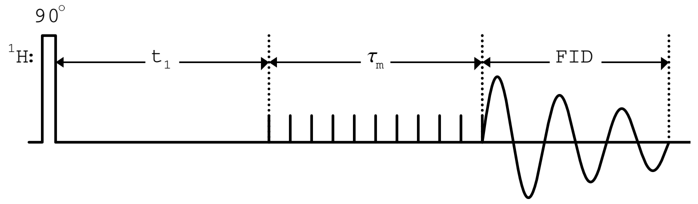

Figure S6 (copy of Figure 2.3). Basic 2D TOCSY.

The only novel element, relative to the preceding sections, is the `lock[taum]` item. As a result,

1. when `inx` reaches 3, in the execution of the `Do` loop under comment 3, `pulseqItem` is set to `lock[taum]`,
2. the statements under comments 4 to 7, have no effect,
3. the statement under comment 8 changes `pulseqItem` to `{lock}[taum]`,
4. the statements under comment 9 has no effect,
5. the statement under comment 10 takes the second case of the `Which`, and sets `currentDuration` to `taum` and unsets `pulseAngle`,
6. the `Do` loop under comment 11 goes through just one cycle, in which
   (a) `currentInnerItem` is set to `lock`,
   (b) `{$udots[], $splock[80]}`is appended to `latent[1]`,
   (c) `alignAndFill[ betweenArrows, {{Mathematica1,10},t,{Courier,10}, subscript,m,basescript},80]`, *i.e.* the expression for τ_m, is appended to `latentX[1]`,
7. `gml` converts the representation of the spin lock, in `latent[1]`, into the graphic objects that draw the line of vertical dots and the comblike depiction. This is produced by the procedure `splock` that is self-explanatory. The graphic objects are included in `activeAll`.

5 Some further details of the displayed output

5.1 Vertical arrangement of channels

The statements under comment 1.1 of `drawPulseSequence` work as follows.

1. The default `displaySequence` is `C13, H1`.
2. The channel specification for the HETCOR example is `ch[H1, C13]`.
3. The body of the `Do` loop is executed with `ii` taking the values 1 then 2.
4. When `ii` is 1,
   (a) `Position[channels, displaySequence[[ii]]]` is `{{2}}`,
   (b) the expression `#=!={}` confirms that this is not null,
   (c) the index 2 is appended to `interim`.
5. When `ii` is 2, the index 1 is appended to `interim` in the same way.
6. `leftOver` is the complement of `{1, 2}` and `{2, 1}`, which is `{}`.
7. Hence `indixes` is set to `{2, 1}`.

In general, `indixes` consists of

1. the elements of `displaySequence` that are present in `channels`, in the order of occurrence in `displaySequence`, followed by
2. the elements of `channels` that are not in `displaySequence`, in the order of occurrence in `channels`.

5.2 Explanatory comments

The statement `writeNotes[`*name*`]` under comment 1 of `drawPulseSequence` writes the LaTeX file `note.`*name*`.tex` containing information about the pulse sequence diagram in particular circumstances. It can then be incorporated above or below the diagram by a LaTeX `\input` command. It is used, in the working document, to provide additional information for Figures 3.2, 3.3, 4.1, 4.2 and 4.3. At present, the first three sections of this procedure support the following kinds of comment.

1. If a gradient pulse occurs during a delay of specified duration, the note is a reminder that this must be long enough for the pulse. The LaTeX string for the delay t has to be specified as the value of `forNote[`t`]`. The current version only includes these values for the examples in the working paper, *i.e.* $t_1/2$, $t_e/2$ and Δ.
2. If a pulse sequence specification mentions N15 (in the `ch` item), the note states $J = J^1_{NH}$, and other hetero atoms are handled correspondingly.
3. The command `let[`u`->`v`]` replaces u by v in a pulse sequence that is being edited. It also associates u and v in `abbreviationList`. This leads to the equation $v = u$ in the comments list.

6 Pulse sequence editing commands

Commands that edit APSEQ representations of pulse sequences are supported by short procedures that are easy to write using a few simple principles. The present version of APSEQ just contains the editing commands that were needed for the examples in the working paper. This repertoire will be extended as need occurs.

6.1 Replacement commands

The action of the MATHSCAPE function `replaceUsing` is shown by

1. `(a+b) // replaceUsing[a->c]` $\longrightarrow$ `c+b`,
2. `(a+b) // replaceUsing[{a->b+c, b-> d}]` $\longrightarrow$ `b+c+d`,

In APSEQ,

1. `replace[`*rule*(*s*)`][`*target*`]` wraps `replaceUsing`,
2. *target*`//replace[`*object*`,`*k*`,by[`*items*`]]` is mnemonic and is just six statements long.

The `changeAtoms` command accommodates the need to change the strength of gradient pulses in proportion to the gyromagnetic ratio.

It is very important to note that users can specify mnemonic commands that are convenient and natural, in their respective working environments, and that can be implemented with almost trivial ease.

6.2 Insertions

Several of the APSEQ editing commands are supported by `insertPulses[{`*insertion*`}, before[`*object*`,`*k*`]]`, where *k* is an integer that defaults to 1. The action is very simple.

1. `objectCursorList` is set to the `Position` of the object in the target pulse sequence.
2. Inconsistent arguments cause an error exit.
3. The builtin MATHEMATICA `Insert` function inserts
4. `sequence[`*insertion*`]` at the appropriate position.
5. The place holder `sequence` is replaced by `Sequence`, which disappears.

The more specific `insertPulses[{`*insertion*`}, after[del[`*d*`], `*k*`]]`, where *k* is an integer, uses the same principles and is even shorter.

The functions to insert spin echoes and spin locks follow the same pattern and are self explanatory.

6.3 Inserting a gradient pulse

This takes a few more statements. The basic non-gradient HSQC of Figure 3.1 is converted to the gradient selected HSQC of Figure3.2 by

```
pulseSeq[hsqc, gradientSelected] =
 pulseSeq[hsqc, basic] //
  pipe[
   insertGradientPulse[gp[4], in[del[t1/2], 2]],
   insertGradientPulse[gp[-1], in[del[1/(4J)], 3]]]
```

This converts

```
{ch[H1,C13], del[1/(4J)], p[180], del[1/(4J)], p[90], del[t1/2], {p[180],},
del[t1/2], p[90], del[1/(4J)],
p[180], del[1/(4J)], {fid, dec}[taq]}
```

to

```
{ch[H1, C13, Gz],...{del, del, gp[4]}[t1/2], p[90],
{del, del, gp[-1]}[1/(4J)],...}
```

where the pieces denoted by ... are unchanged. The insertion function

1. tests if the target contained a gradient channel — this would have been shown by `Gz` as the final item of the `ch` item,
2. inserts `Gz` if absent, by appending `Gz` to the old `ch` item (which has to be the first) and prepending the new `ch` item to the rest of the old pulse sequence,
3. sets `nonGzCount` to the number of channels besides the gradient chann el,
4. replaces the `del[`*duration*`]` by `{del,..., del, gp[`*strength*`]}`, where the ... stand for as many `del` items as needed (in this case, none).

Action is focused on the appropriate `del` item in the old pulse by the MATHSCAPE targeting expression `to[del[d][k]]`.

7 Adapting an assignment

The `adapt[last[statement], by[`*action*`]]` and related commands all

1. retrieve the history of the session, using `InString /@ Range[$Line-1]`, as a list of strings,
2. find the appropriate element of this list, searching backwards with an appropriate key,
3. apply `Hold`, if necessary, to the corresponding expression,
4. edit appropriately.