3 Pulse Waveform

3.1 Overview

The class ***PulWaveform*** embodies a pulse waveform in NMR simulations. The class maintains the pulse waveform as a discrete function whose individual points (waveform steps) consist of three values, $\{\gamma B_1, \phi, t_p\}$, the rf-field strength, the rf-field phase, and the time the rf-field is applied. This class works with other GAMMA classes for handling shaped pulses, composite pulses, & pulse cycles.

Waveform in GAMMA Pulse Hierarchy

Waveform **Composite/Shaped Pulse** **Pulse Cycle**

step	γB_1	ϕ	time
1	2.5	0	0.1
2	2.5	90	0.2
3	2.5	0	0.1

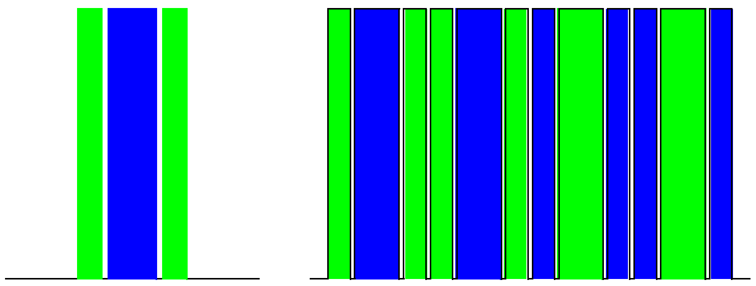


Figure 3-1 A waveform (e.g. composite 180) is just a list of pulse steps, each step having a specified rf-field strength, rf-field phase, and length (e.g. shown in kHz and msec). The waveform is used to produce a shaped or composite pulse which can be used as a step in an NMR pulse sequence. In turn, the composite pulse can be used to form a pulse cycle (repeated pulses with phase changes, e.g. MLEV-4) which may also be used during NMR simulations. This hierarchy continues to accomodate supercycles and cycles with interdispersed pulses.

3.2 Chapter Contents

3.2.1 Pulse Waveform Functions

3.3 Construction & Assignment

3.3.1 PulWaveform

Usage:

```
#include <PulWaveform.h>
PulWaveform()
PulWaveform(row_vector& wfsteps, row_vector& wftimes, const String& wfname, int wfrad=0)
PulWaveform(const PulWaveform& PWF)
```

Description:

The function ***PulWaveform*** is used to create a pulse waveform in GAMMA.

1. PulWaveform() - Creates an “empty” NULL pulse waveform. Can be later filled by an assignment.
2. PulWaveform(row_vector& wfsteps, row_vector& wftimes, const String& wfname, int wfrad=0) - Creates a new waveform named ***wfname*** whose step rf-amplitudes and phase are contained in the input vector ***wfsteps*** and whose step lengths are in the vector wftimes. The flag ***wfrad*** indicates whether the input phases are in degrees (default) or radians.
3. PulWaveform(const PulWaveform& PWF) - Called with another pulse waveform quantity this function constructs an identical waveform equal to the input ***PWF***.

Return Value:

PulWaveform returns no parameters. It is used strictly to create an pulse waveform.

Examples:

```
#include <PulWaveform.h>
PulWaveform PWF;                    // Empty pulse waveform.
int nsteps = 100;
row_vector gB1s = Lorentzian(
row_vector times(nsteps, 0.001);
PWF1(gB1s, times, “new”);           // Here is a mock Lorentzian pulse waveform
PulWaveform PWF2(PWF1);             // An identical copy of PWF1
```

See Also: =

3.3.2 =

Usage:

```
#include <PulWaveform.h>
void PulWaveform operator = (PulWaveform &PWF)
```

Description:

The unary ***operator =*** (the assignment operator) allows for the setting of one pulse waveform equal to another pulse waveform. The waveform being assigned to will be overwritten by ***PWF***.

Return Value:

None, the function is void

Examples:

```
#include <PulWaveform.h>
#include <PulWALTZ.h>
PulWaveform PWF;                    // Empty pulse waveform.
PWF = WF_WALTZQ(875.0);             // Set PWF to WALTZ-Q waveform @ 875 Hz
```

See Also: PulWaveform

3.4 Access Functions

3.4.1 steps

Usage:

```
#include <PulWaveform.h>
int PulWaveform::steps( )
double PulWaveform::steps(double td)
```

Description:

The function ***steps*** with no arguments returns the number of individual steps defined in the pulse waveform. Of a length ***td*** is given, the function returns the number of steps that will span the length specified.

Return Value:

The function returns either an integer or a double.

Example:

```
#include <PulWaveform.h>
#include <PulMLEV.h>
MLEV MP(1260.0, “1H”);                    // Set up MLEV parameters
PulWaveform PWF = MP.WF();                // Set default MLEV waveform (comp. 180)
cout << “\nMLEV Waveform has “            // Output the number of steps
      << PWF.steps() << “ steps.”;
double td = 0.03;                         // Set a delay time
cout << “\nTo span “ << td << “ seconds “ // Output waveforms spanning delay
      << “requires “ << PWF.steps(td)
      << “ waveforms”;
```

See Also:

3.4.2 name

Usage:

```
#include <PulWaveform.h>
String PulWaveform::name( )
```

Description:

The function ***name*** returns the name of the pulse waveform.

Return Value:

The function returns a string.

Example:

```
#include <PulWaveform.h>
#include <PulGARP.h>
GARP GP(1260.0, “1H”);                      // Set up GARP parameters
PulWaveform PWF = GP.WF();                  // Set default GARP waveform (25 steps)
cout << “\nWorking with “ << PWF.name()  // Output the waveform name
        << “ waveform.”;
```

3.4.3 values

Usage:

```
#include <PulWaveform.h>
row_vector PulWaveform::values( )
```

Description:

The function ***values*** returns a row_vector containing values which define the pulse waveform steps. The ith vector value contains the values {γB1, ϕ}, where the real component is the rf-field strength in Hz, and the imaginary component is the rf-phase in degrees (or radians).

Return Value:

The function returns a row vector.

Example:

```
#include <PulWaveform.h>
#include <PulGARP.h>
GARP GP(1260.0, “1H”);                      // Set up GARP parameters
PulWaveform PWF = GP.WF();                  // Set default GARP waveform (25 steps)
row_vector vals = PWF.values();             // Get array of strengths and phases
```

See Also:

3.4.4 lengths

Usage:

```
#include <PulWaveform.h>
row_vector PulWaveform::lengths( )
```

Description:

The function ***lengths*** returns a row_vector containing values which define the pulse waveform step lengths. The real part of the ith vector value contains the length os step i in seconds.

Return Value:

The function returns a row vector.

Example:

```
#include <PulWaveform.h>
#include <PulGARP.h>
GARP GP(1260.0, “1H”);                // Set up GARP parameters
PulWaveform PWF = GP.WF();            // Set default GARP waveform (25 steps)
row_vector ts = PWF.lengths();        // Get array of lengths
```

See Also:

3.4.5 value

Usage:

```
#include <PulWaveform.h>
complex PulWaveform::value(int i)
```

Description:

The function ***value*** returns a compex number for the values which define the pulse waveform step ***i***. The value contains the number {γB1, φ}, where the real component is the rf-field strength in Hz, and the imaginary component is the rf-phase in degrees (or radians).

Return Value:

The function returns a complex number.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.4.6 phase

Usage:

```
#include <PulWaveform.h>
double PulWaveform::phase(int i)
```

Description:

The function ***phase*** returns the value of the rf-field phase at pulse waveform step ***i*** in degrees (or radians).

Return Value:

The function returns a double.

3.4.7 strength

Usage:

```
#include <PulWaveform.h>
double PulWaveform::strength(int i)
```

Description:

The function ***strength*** returns the value of the rf-field amplitude at pulse waveform step ***i*** in Hz.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.4.8 length

Usage:

```
#include <PulWaveform.h>
double PulWaveform::length(int i)
```

Description:

The function ***length*** returns the length of th pulse waveform in seconds.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.5 Auxiliary Functions

3.5.1 maxlength

Usage:

```
#include <PulWaveform.h>
double PulWaveform::maxlength( )
```

Description:

The function ***maxlength*** returns the length of the longest waveform step.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.5.2 minlength

Usage:

```
#include <PulWaveform.h>
double PulWaveform::maxlength( double cutoff=1.e-13)
```

Description:

The function ***minlength*** returns the length of the shortest non-zero waveform step. The step is considered to be of zero length if it falls below the value set by ***cutoff***.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.5.3 gamB1const

Usage:

```
#include <PulWaveform.h>
int PulWaveform::gamB1const( )
```

Description:

The function ***gamB1const*** returns true if all steps in the waveform have the same rf-field strength.

Return Value:

The function returns an integer.

Example:

```
#include <PulWaveform.h>
```

See Also: phaseconst, timeconst

3.5.4 phaseconst

Usage:

```
#include <PulWaveform.h>
int PulWaveform::phaseconst( )
```

Description:

The function ***phaseconst*** returns true if all steps in the waveform have the same rf-field phase.

Return Value:

The function returns a string.

Example:

See Also: timeconst, gamB1const

3.5.5 timeconst

Usage:

```
#include <PulWaveform.h>
int PulWaveform::timeconst( )
```

Description:

The function ***timeconst*** returns true if all steps in the waveform have the same length.

Return Value:

The function returns a row vector.

Example:

```
#include <PulWaveform.h>
```

See Also: phaseconst, gamB1const

3.5.6 WFs

Usage:

```
#include <PulWaveform.h>
double PulWaveform::WFs(double td)
```

Description:

The function ***WFs*** returns the number of pulse waveforms needed to span the length ***td***.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.5.7 fullWFs

Usage:

```
#include <PulWaveform.h>
int PulWaveform::fullWFs(double td)
```

Description:

The function ***fullWFs*** returns the number of complete pulse waveforms that fit within the time span ***td***.

Return Value:

The function returns an integer.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.5.8 fullsteps

Usage:

```
#include <PulWaveform.h>
int PulWaveform::fullsteps(double td)
```

Description:

The function ***fullsteps*** returns the number of complete pulse waveform steps that fit within the time span ***td***.

Return Value:

The function returns an integer.

Example:

```
#include <PulWaveform.h>
```

3.5.9 sumlength

Usage:

```
#include <PulWaveform.h>
double PulWaveform::sumlength(int i)
```

Description:

The function ***sumlength*** returns the summed length over the first ***i*** steps of the waveform.

Return Value:

The function returns a double.

Example:

```
#include <PulWaveform.h>
```

3.6 Plotting Functions

3.6.1 GP

Usage:

```
#include <PulWaveform.h>
void PulWaveform::GP(int type=0, int split=0, int ends=0, int N=1)
```

Description:

The function ***GP*** will produce a plot of the waveform on screen using the ***Gnuplot*** program if available. The function will plot either the rf-intensity versus time (***type = 1***) or the rf-phase versus time (***type = 0***). Individual waveforms steps will be separated by ***split*** multiples of one tenth the first waveform step length. Ends will be drawn on the plot of length ends*length of first pulse step. There will be ***N*** waveforms plotted.

Return Value:

Void. Aplot is produced on screen if Gnuplot is available.

Example:

```
#include <PulWaveform.h>
#include <PulMLEV.h>
MLEV MP(1260.0, “1H”);                 // Set up MLEV parameters
PulWaveform PWF = MP.WF();             // Set default MLEV waveform (comp. 180)
PWF.GP(1, 1, 1);                       // Make strength vs. time plot
```

See Also: FM

3.6.2 FM

Usage:

```
#include <PulWaveform.h>
void PulWaveform::FM(int type=0, int split=0, int ends=0, int N=1)
```

Description:

The function ***FM*** will produce a plot of the waveform in FrameMaker MIF format. The function will plot either the rf-intensity versus time (***type = 1***) or the rf-phase versus time (***type = 0***). Individual waveforms steps will be separated by ***split*** multiples of one tenth the first waveform step length. Ends will be drawn on the plot of length ends*length of first pulse step. There will be ***N*** waveforms plotted. The output filename will reflect the name of the waveform, the type plotted, and have a mif suffix.

Return Value:

Example:

```
#include <PulWaveform.h>
#include <PulMLEV.h>
MLEV MP(1260.0, “1H”);                 // Set up MLEV parameters
PulWaveform PWF = MP.WF();             // Set default MLEV waveform (comp. 180)
```

```
PWF.FM(1, 1, 1);                    // Make strength vs. time plot
```

See Also: GP

3.7 Input/Output Functions

3.7.1 printBase

Usage:

```
#include <PulWaveform.h>
ostr PulWaveform::printBase(ostream& ostr)
```

Description:

The function ***printBase*** will put basic information regarding the waveform into the output stream ***ostr*** given as an input argument

Return Value:

The function modifies the output stream and returns it.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.7.2 printSteps

Usage:

```
#include <PulWaveform.h>
ostr PulWaveform::printSteps(ostream& ostr)
```

Description:

The function ***printSteps*** will put information regarding the pulse cycle individual step phases into the output stream ***ostr*** given as an input argument.

Return Value:

The function modifies the output stream and returns it.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.7.3 print

Usage:

```
#include <PulWaveform.h>
ostr PulWaveform::print(ostream& ostr, int full=0)
```

Description:

The function ***print*** will put information regarding the pulse waveform into the output stream ***ostr*** given as an input argument. If the optional flag ***full*** has been set to non-zero, information regarding individual pulse waveform steps will also be added (which can be a lot data) added to the output stream.

Return Value:

The function modifies the output stream and returns it.

Example:

```
#include <PulWaveform.h>
```

See Also:

3.7.4 <<

Usage:

```
#include <PulWaveform.h>
ostream& operator << (ostream& ostr, PulWaveform& PWF)
```

Description:

The operator **<<** adds the pulse waveform specified as an argument ***PWF*** to the output stream ***ostr***.

Return Value:

None.

Example(s):

```
#include <PulWaveform.h>
```

See Also:

3.8 Description

3.8.1 Introduction

Class ***PulWaveform*** is designed to faclitate the use of generic shaped pulses, composite pulses, and pulse trains in GAMMA. There are a wide variety of such waveforms commonly used in modern NMR spectroscopy. In GAMMA, as in an NMR experiment, we should like to use pulses and pulse trains generated from arbitrary waveforms as individual steps in a general pulse sequence. This includes use in variable delays as part of multi-dimensional experiements and/or use in continuous pulse trains during acquisition steps.

3.8.2 Pulse Waveform Basis

We consider a pulse waveform as involving four basic features: 1.) The ***# steps***, 2.) The ***rf-field strength*** of each step, the ***rf-phase*** of each step, the ***length*** of each step: N, $\{\gamma B_1, \phi, t_p\}$. One example would be a Gaussian pulse waveform. In this case the Gaussian function is broken up into N steps each having the same phase and length but with varying field strength.

Gaussian Waveform

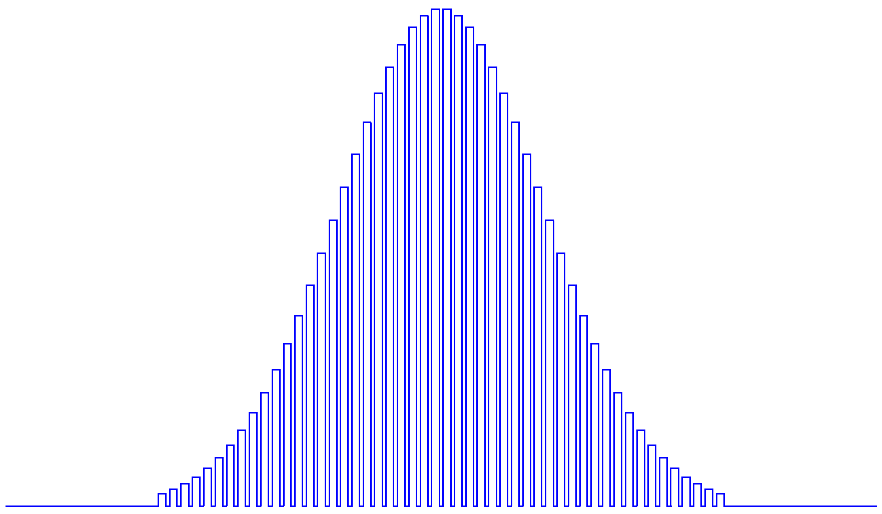


Figure 3-2 A Gaussian waveform in GAMMA. Each step here has the same length and phase whereas the rf intensity changes according the the Gaussian function.This figure was made by the program GaussWF.cc on page 55 of this chapter.

A Gaussian pulse waveform, such as shown above, simply maintains the steps and rf-amplitudes, in this case the step length is constant. As an alternative example, the we can consider a GARP-1 25-step composite pulse as our waveform. In this case the rf-field strength is maintained constant whereas the phase and length of the steps (individual pulses) change.

Basic GARP 25 Step Sequence

Step	Angle	Step	Angle	Step	Angle
1	30.5	9	134.5	17	258.4
2	$\overline{55.2}$	10	$\overline{256.1}$	18	$\overline{64.9}$
3	257.8	11	66.4	19	70.9
4	$\overline{268.3}$	12	$\overline{45.9}$	20	$\overline{77.2}$
5	69.3	13	25.5	21	98.2
6	$\overline{62.2}$	14	$\overline{72.7}$	22	$\overline{133.6}$
7	85.0	15	119.5	23	255.9
8	$\overline{91.8}$	16	$\overline{138.2}$	24	$\overline{65.6}$
				25	$\overline{53.4}$

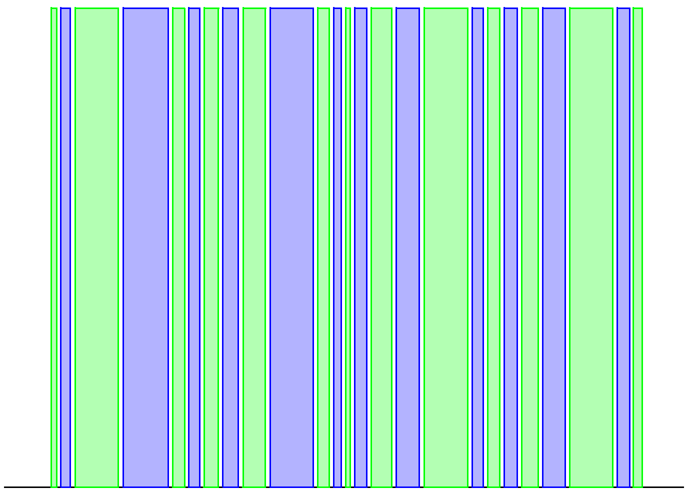


Figure 3-3 The basic 25-step GARP waveform. The blue steps indicate pulse that are applied with a 180 degree phase shift, as indicate by a bar in the table listing. The program which produced this plot can be found at the end of this chapter, GarpWF0.cc on page 55.

See the DANTE module for an example of a waveform with pulse and delay steps mixed and the CHIRP module should contain a nice example of a constantly changing (phase and intensity) waveform.

3.8.3 Pulse Waveform Construction

The examples in the previous section generated and plotted waveforms which are supplied through pre-existing functions available in GAMMA. Users are free to look through the source code of the individual modules containing the function(s) to see exactly how they were generated. In this section it is shown explicitly how you can build your own pulse waveforms.

Building a waveform requires two arrays (row-vectors) that have length equal to the intended waveform steps. The first vector will contain the rf strengths and phases and the second will contain the step lengths. An optional name can be associated with the waveform too. Let us just make the basic GARP 25-step waveform given in a previous figure. First, here is how to make the vector of amplitudes and phase:

```
row_vector WFsteps(25);                          // Vector for waveform
double gamB1 = 2000;                             // Field strength to 2 kHz
double phi = 0.0;                                // Base phase
double phibar = phi + 180.0;                     // Alternate phase
WFsteps.put(complex(gamB1,phi),      0);         // Set waveform values
WFsteps.put(complex(gamB1,phibar), 1);           //  { gamB1, phi }
WFsteps.put(complex(gamB1,phi),     2);
WFsteps.put(complex(gamB1,phibar), 3);
WFsteps.put(complex(gamB1,phi),     4);
WFsteps.put(complex(gamB1,phibar), 5);
WFsteps.put(complex(gamB1,phi),     6);
WFsteps.put(complex(gamB1,phibar), 7);
WFsteps.put(complex(gamB1,phi),     8);
WFsteps.put(complex(gamB1,phibar), 9);
WFsteps.put(complex(gamB1,phi),   10);
WFsteps.put(complex(gamB1,phibar),11);
WFsteps.put(complex(gamB1,phi),    12);
WFsteps.put(complex(gamB1,phibar),13);
WFsteps.put(complex(gamB1,phi),    14);
WFsteps.put(complex(gamB1,phibar),15);
WFsteps.put(complex(gamB1,phi),    16);
WFsteps.put(complex(gamB1,phibar),17);
WFsteps.put(complex(gamB1,phi),    18);
WFsteps.put(complex(gamB1,phibar),19);
WFsteps.put(complex(gamB1,phi),    20);
WFsteps.put(complex(gamB1,phibar),21);
WFsteps.put(complex(gamB1,phi),    22);
WFsteps.put(complex(gamB1,phibar),23);
WFsteps.put(complex(gamB1,phi),    24);
```

Next we make a vector of step (pulse) lengths. These will be adjusted to produce the proper pulse angles for GARP:

```
row_vector WFtimes(25);                         // Vector for step times
double tdegree = 0;                             // Increment time per pulse
if(gamB1>0) tdegree = 1/(gamB1*360);            // degree
WFtimes.put(30.5*tdegree,  0);
WFtimes.put(55.2*tdegree,  1);
WFtimes.put(257.8*tdegree, 2);
WFtimes.put(268.3*tdegree, 3);
WFtimes.put(69.3*tdegree,  4);
WFtimes.put(62.2*tdegree,  5);
WFtimes.put(85.0*tdegree,  6);
WFtimes.put(91.8*tdegree,  7);
WFtimes.put(134.5*tdegree, 8);
WFtimes.put(256.1*tdegree, 9);
WFtimes.put(66.4*tdegree,  10);
WFtimes.put(45.9*tdegree,  11);
WFtimes.put(25.5*tdegree,  12);
WFtimes.put(72.7*tdegree,  13);
WFtimes.put(119.5*tdegree, 14);
WFtimes.put(138.2*tdegree, 15);
WFtimes.put(258.4*tdegree, 16);
WFtimes.put(64.9*tdegree,  17);
WFtimes.put(70.9*tdegree,  18);
WFtimes.put(77.2*tdegree,  19);
WFtimes.put(98.2*tdegree,  20);
WFtimes.put(133.6*tdegree, 21);
WFtimes.put(255.9*tdegree, 22);
WFtimes.put(65.6*tdegree,  23);
WFtimes.put(53.4*tdegree,  24);
```

Now we can make our waveform:

```
PulWaveform GWF(WFsteps, WFtimes, “GARP-1”);
```

Now that we have a waveform we can make composite pulses and pulse cycles which can be used as single steps, mixing steps, or during acquisition steps during NMR simulations. Remember, the waveform itself is just a container for shaped/composite pulse information. In itself it doesn’t do any calculations.

3.8.4 Pulse Waveform Utility

In GAMMA, pulse waveforms have little functionality. The class exists largely to serve higher level pulse entities such as composite pulses and pulse cycle. In effect, one builds the pulses required in NMR simulations from pulse waveforms. To illustrate this, consider a simulation in which CHIRP decoupling is desired during acquisitions. The basic CHIRP waveform is shown in the following figure.

Typical Chirp Sequence Amplitude and Phase

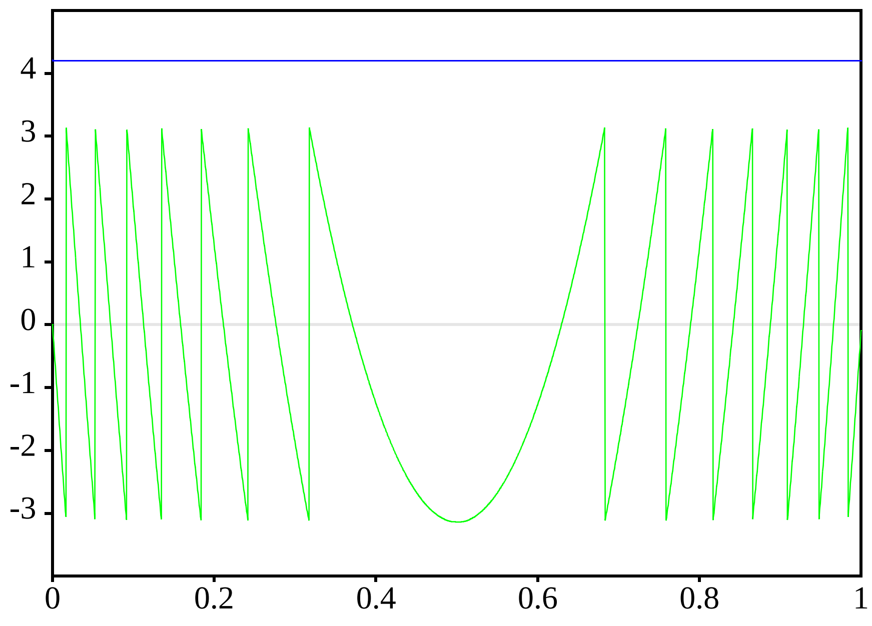


Figure 3-4 The rf amplitude and phase for a typical Chirp. The amplitide is kept constant at 4.2 kHz (scaled down by 1000 to fit on the plot) and ploted in blue. The phase is shown plotted in green and expressed in units of radians. The phase range is shown within the limits $phase \ni [-\pi,\pi]$. The program which produced the plot can be found in the GAMMA CHIRP documentation.

The previous section detailed how one might produce the above CHIRP waveform in GAMMA. To actually use a pulse based on this waveform ***demands*** more information than what is contained in the waveform itself. The computation must have knowledge of the spin system evolving, the channel the pulse will be applied on, possibly even dynamical and exchange parameters. By joining a spin system with a CHIRP pulse waveform we can produce a **composite pulse** that can indeed evolve the system under a CHIRP pulse. Here is some example GAMMA code:

```
spin_system sys;                          // Declare a spin system
sys.read("filein.sys");                   // Read in the spin system
PulWaveform WF = WF_CHIRP95(4200);        // Set up a CHIRP-95 waveform
PulComposite PC(WF, sys, "13C");          // Composite pulse on 13C channel
```

We could easily use repeated CHIRP composite pulses to evolve during a t1 evolution or during an acquisition. But, as is typical in using such decoupling sequences, the base waveform is phase cycled when repeatedly applied. In fact, CHIRP-95 is defined to be the base waveform cycled with a 5 step Tyko-Pines phase change and they supercycled in typical MLEV-16 fashion. Thus, if the CHIRP waveform is broken up into 200 steps the full CHIRP-95 decoupling sequence will be 16,000 steps long per each cycle. To perform a simulated acquisiton during such a process demands that the program is very careful in timing acquisition points relative to position in the cycle. In additon it is somewhat important to reutilize mathmatical entities which are repeatedly called for due to the nature of the sequence's symmetry. In GAMMA, these factors are automatically taken into account with "higher" pulse classes: PulCycle and PulSupCycle.

Relationship Between Chirp, Chirp Cycle, and Chirp Supercycle

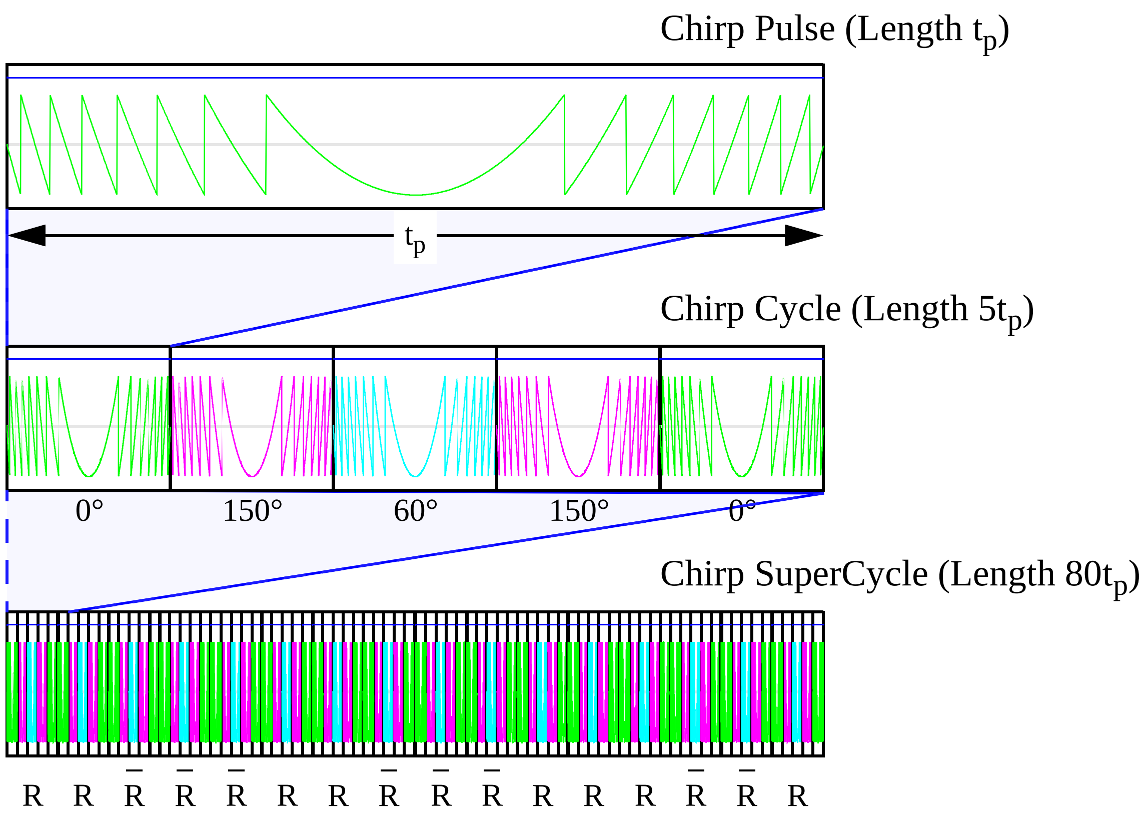

Chirp Pulse (Length t_p)
t_p
Chirp Cycle (Length $5t_p$)
0°
150°
60°
150°
0°
Chirp SuperCycle (Length $80t_p$)
$R \quad R \quad \bar{R} \quad \bar{R} \quad \bar{R} \quad R \quad R \quad \bar{R} \quad \bar{R} \quad \bar{R} \quad R \quad R \quad R \quad \bar{R} \quad \bar{R} \quad R$

Figure 3-5 The relationship between the Chirp supercycle used for broadband decoupling and the basic Chirp sequence. The Chirp cycle is that developed by Tyko and Pines and consists 5 Chirps in succession with relative phase changes. The Chirp supercycle contains 16 of the Chirp 5-step cycles, each of which has a relative phase of zero (R) or 180 (R with a bar). The supercycle is MLEV-16 containing where the 5-step Chirp cycle is the primary unit. Thus there are 80 Chirp units in 1 supercycle, and the supercycle is repeated during decoupling.

For clarity lets summarize. Any ***waveform*** can be declared in GAMMA (e.g. CHIRP). To apply a pulse associated with the waveform one uses a ***composite pulse*** built from the waveform (or provided by a GAMMA function). If the pulse is to be repeated with a phase cycle one uses a ***pulse cycle*** made from the composite pulse. If the pulse cycle is itself cycled then one uses a ***pulse supercycle*** which is made from the pulse cycle.

3.9 Chapter Source Codes

GaussWF.cc

```
/* GaussWF.cc ********************************************************************
**                                                                              **
**                GAMMA Pulse Waveform Example Program                          **
**                                                                              **
**  This program uses the class PulWaveform and the Gaussian pulse              **
**  module to build a Gaussian pulse waveform.  It doesn't do anything          **
**  with the waveform but spit out a waveform plot to the screen using          **
**  Gnuplot and make a FrameMaker MIF file of the plot.  The plot is            **
**  set for time vs rf-amplitude.                                               **
**                                                                              **
** Author:      S.A. Smith                                                      **
** Date:        3/9/98                                                          **
** Update:      3/9/98                                                          **
** Version:     3.5.4                                                           **
** Copyright:   S. Smith.  You can modify this program as you see fit           **
**              for personal use, but you must leave the program intact         **
**              if you re-distribute it.                                        **
**                                                                              **
**********************************************************************************/

#include <gamma.h>

main()
  {
  PulWaveform PW = WF_Gaussian(600, 0.01, 50);
  PW.GP(1, 5, 20);
  PW.FM(1, 5, 20);
  cout << "\n";
  cout.flush();
  }
```

GarpWF0.cc

```
/* GarpWF0.cc *******************************************************************  **
**                                                                               **
**                 GAMMA GARP Simulation Example Program                         **
**                                                                               **
** This program examines the basic GARP sequence Waveform.  I does no            **
** NMR computations involving GARP, it merely spits out plots so that            **
** the default GARP (GARP-1) sequence can be readily viewed.                     **
**                                                                               **
** Assuming a.out is the executable of this program, then the following          **
** command will generate a single 25 step GARP-1 waveform which will             **
** be displayed on screen if Gnuplot is available.  It will also make            **
** an editable FrameMaker MIF file of the waveform.                              **
**                                                                               **
**                          a.out                                                **
**                                                                               **
** Author:    S.A. Smith                                                         **
** Date:      2/27/98                                                            **
** Copyright: S.A. Smith, February 1998                                          **
**                                                                               **
***********************************************************************************/

#include <gamma.h>                                  // Include GAMMA

main(int argc, char* argv[])
  {
  cout << "\n\n\t\t\tGAMMA GARP Waveform Program 0\n";
  GARP GP(500.0, "1H");                             // Set GARP parameters
  PulWaveform PWF = GP.WF_GARP1();                  // Construct waveform
  PWF.GP(1, 1, 5);                                  // Plot waveform(s), gnuplot
  PWF.FM(1, 1, 5);                                  // Plot waveform(s), Framemaker
  cout << "\n\n";                                   // Keep screen nice
  }
```

4 Gaussian Pulses

4.1 Gaussian Pulse Sections

4.2 Gaussian Pulse Functions

4.3 Gaussian Pulse Figures & Tables

4.4 Gaussian Pulse Programs

4.5 Gaussian Pulse Propagators

4.5.1 Gaussian

Usage:

```
#include <P_Gaussian.h>
gen_op Gaussian(spin_system& sys, gen_op& H, String& Iso,
                                                  double td, double theta, double phi=0.0)
void Acquire1D(gen_op &Op, gen_op &H, double dt=0)
void Acquire1D(gen_op &Op, super_op &L, double dt=0)
void Acquire1D(const Acquire1D &ACQ1)
```

Description:

The function *Acquire1D* is used to create a 1-dimensional acquisition computational core.

1. Acquire1D() - Creates an “empty” NULL acquire1D. Can be later filled by an assignment.
2. Acquire1D(gen_op &Op, gen_op &H, double dt) - Called with the operator for which the expectation values are desired (Op) and the static Hamiltonian (H) under which the system density operator will evolve, this function constructs an appropriate acquire1D for future computation use. The optional value dt may be set for an incremental delay time. This produces an exponential Liouvillian for rapid generation of time domain spectra.
3. Acquire1D(gen_op &Op, super_op &L, double dt) - Called with the operator for which the expectation values are desired (Op) and the system Liouvillian (L) under which the system density operator will evolve, this function constructs an appropriate acquire1D for future computation use. The optional value dt may be set for an incremental delay time. This produces an exponential Liouvillian for rapid generation of time domain spectra.
4. Acquire1D(const Acquire1D &ACQ1) - Called with another acquire1D quantity this function constructs an identical acquire1D to the input ACQ1.

Return Value:

Acquire1D returns no parameters. It is used strictly to create an acquire1D.

Examples:

See Also: =

4.5.2 =

Usage:

```
#include <P_Gaussian.h>
void acquire1D operator = (acquire1D &ACQ1)
```

Description:

The unary operator = (the assignment operator) allows for the setting of one acquire1D to another acquire1D. If the acquire1D being assigned to exists it will be overwritten by the assigned acquire1D.

Return Value:

None, the function is void

Examples:

See Also: acquire1D

4.5.3 Gpulse_Hs

Usage:

```
#include <P_Gaussian.h>
void Gpulse_Hs(gen_op* Hs, gen_op& Ho, gen_op& Fxy,
                                        int N, double ang, double tp, double fact)
```

Description:

The function ***Gpulse_Hs*** generates a series of active Hamiltonians applicable to a Gaussian shaped pulse. The Hamiltonians are defined only in the rotating frame of the rf-field of the pulse. The array of general operators ***Hs*** is filled with ***N*** operators representing the Gaussian waveform. The waveform consists of ***N*** steps, is of length ***tp*** (sec.), begins at intensity ***fact*** (decimal% of maximum), and will rotate magnetization on resonance by angle ***ang*** (degrees). The operator ***Fxy*** sets the pulse phase and selectivity. The operator ***Ho*** is the isotropic system Hamiltonian.

Return Value:

The function is void, it will fill operator array Hs.

Example:

```
#include <P_Gaussian.h>
Acquire1D ACQ1(det,L,sigmaeq);                    // Set up for 1D acquisition
```

See Also:

4.5.4 Gpulse_Us

Usage:

```
#include <P_Gaussian.h>
void Gpulse_Us(gen_op* Us, gen_op& Ho, gen_op& Fxy,
                                        int N, double ang, double tp, double fact)
```

Description:

The function ***Gpulse_Us*** generates a series of propagators applicable to a Gaussian shaped pulse. The propagators are defined only in the rotating frame of the rf-field of the pulse. The array of general operators ***Us*** is filled with ***N*** propagators representing the Gaussian waveform. The waveform consists of ***N*** steps, is of length ***tp*** (sec.), begins at intensity ***fact*** (decimal% of maximum), and will rotate magnetization on resonance by angle ***ang*** (degrees). The operator ***Fxy*** sets the pulse phase and selectivity. The operator ***Ho*** is the isotropic system Hamiltonian.

Return Value:

The function is void, it will fill operator array Hs.

Example:

```
#include <P_Gaussian.h>
Acquire1D ACQ1(det,L,sigmaeq);                    // Set up for 1D acquisition
```

See Also:

4.5.5 Gpulse_U

Usage:

```
#include <P_Gaussian.h>
gen_op Gpulse_U(gen_op& Ho, gen_op& Fxy,
                                   int N, double ang, double tp, double fact)
```

Description:

The function ***Gpulse_U*** generates a propagator which will evolve a spin system under a Gaussian shaped pulse. The waveform consists of ***N*** steps, is of length ***tp*** (sec.), begins at intensity ***fact*** (decimal% of maximum), and will rotate magnetization on resonance by angle ***ang*** (degrees). The operator ***Fxy*** sets the pulse phase and selectivity. The operator ***Ho*** is the isotropic system Hamiltonian.

Return Value:

The function a single propagator (operator).

Example:

```
#include <P_Gaussian.h>
Acquire1D ACQ1(det,L,sigmaeq);                    // Set up for 1D acquisition
```

See Also:

4.6 Auxiliary Functions

4.6.1 Gangle

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees. The Gaussian waveform is based on the input field strength ***gamB1***in Hertz, the pulse length ***tau*** in seconds, the number of steps ***N***, and the decimal percent of maximum ***fact*** at the end points.

Return Value:

The function returns a double precision number.

Example:

```
#include <P_Gaussian.h>
double gamB1 = 50;                  // Set gamma*B1 to 50 Hz
int nstp = 1000;                    // Set number of steps to 500
double tp = 0.30;                   // Set the pulse length to 30 ms
cout << "\nPulse Angle: "           // Output the results of Gangle
     << Gangle(gamB1,tp,nstp)       // (roughly 270 degrees in this case)
     << " Degrees";
```

See Also:

4.6.2 GgamB1

Usage:

```
#include <P_Gaussian.h>
double GgamB1 (double angle, double tau, int N, double fact=0.025)
```

Description:

The function ***GgamB1*** returns the Gaussian shaped pulse rf-strength needed to attain an on-resonance rotation angle of ***angle*** degrees. The Gaussian waveform is based on the rotation angle ***angle*** in degrees, the pulse length ***tau*** in seconds, the number of steps ***N***, and the decimal percent of maximum ***fact*** at the end points.

Return Value:

The function returns a double precision number.

Example:

```
#include <P_Gaussian.h>
```

```
double tp = 0.01;                         // Set pulse length to 10 ms
int N = 1001;                             // Set number of steps to 1001
double ang = 270.;                        // Set the angle to 270 degrees
double fact = 0.025;                      // Set cutoff to 2.5% at endpoints
double GB1 = GgamB1(ang,tp,N,fact);       // Get the needed field strength
cout << “\nSet Field to “ << GB1 << “ Hz”; // Output calculated field (~160Hz)
```

See Also:

4.6.3 Gtime

Usage:

```
#include <P_Gaussian.h>
double Gangle(double angle, double gamB1, int N, double fact=0.025)
```

Description:

The function ***Gtime*** returns the Gaussian shaped pulse length in seconds required to attain an on-resonance rotation angle of ***angle*** degrees. The Gaussian waveform is based on the rotation angle ***angle*** in degrees, the input field strength ***gamB1***in Hertz, the number of steps ***N***, and the decimal percent of maximum ***fact*** at the end points.

in degrees.

Return Value:

The function returns a double precision number.

Example:

```
#include <P_Gaussian.h>
double angle = 270.;                      // Set pulse angle to 270 degrees
int nstp = 1001;                          // Set number of steps to 1001
double gamB1 = 50.;                       // Set the field strength to 50 Hz
cout << “\nPulse Length: “                // Output the results of Gtime
     << Gtime(ang,gamB1,nstp)             // (roughly 32 ms in this case)
     << “ Seconds”;
```

See Also:

4.6.4 GNvect

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.6.5 Gvect

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.6.6 GIntvec

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.7 Input/Output Functions

4.7.1 Ghistogram

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.7.2 ask_Gpulse

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.7.3 read_Gpulse

Usage:

```
#include <P_Gaussian.h>
double Gangle(double gamB1, double tau, int N, double fact=0.025)
```

Description:

The function ***Gangle*** returns the Gaussian shaped pulse rotation angle on resonance in degrees.

Return Value:

The function is void, it will alter the input data block.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.7.4 <<

Usage:

```
#include <P_Gaussian.h>
ostream& operator << (ostream& ostr, acquire1D& ACQ1)
```

Description:

The operator << adds the acquisition specified as an argument ***ACQ1*** to the output stream ***ostr***. The format will as follows:

# non-zero points out of # possible

Dwell time: # (if available)

A[i], B[i] pairs

Hilbert space basis.

Return Value:

None.

Example(s):

```
#include <P_Gaussian.h>
```

See Also:

4.8 Auxiliary Functions

4.8.1 size

Usage:

```
#include <P_Gaussian.h>
int acquire1D::size( )
```

Description:

The function ***size*** returns the current dimension over which the index ***p*** will sum in the generalized ***class acquire1D*** acquisition equation

$$\langle Op(t_k)\rangle = \sum_{p}^{size} A_p[B_p]^k$$

Return Value:

The function returns an integer.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.8.2 size

Usage:

```
#include <P_Gaussian.h>
int acquire1D::size( )
```

Description:

The function ***size*** returns the current dimension over which the index ***p*** will sum in the generalized ***class acquire1D*** acquisition equation

$$\langle Op(t_k)\rangle = \sum_{p}^{size} A_p[B_p]^k$$

Return Value:

The function returns an integer.

Example:

```
#include <P_Gaussian.h>
```

See Also:

4.9 Description

4.9.1 Introduction

The module ***P_Gaussian*** provides functions which pertain to Gaussian shaped pulses in NMR simulations. These functions either return propagators which evolve the density operator or they act on the density operator directly.

4.9.2 Analog Mathematical Basis

The Gaussian function is formally given by

$$G(t) = e^{[-(t-t_0)^2/2\sigma^2]} \tag{19-1}$$

where σ is the standard deviation and relates the linewidth at half-height by the relationship

$$t_{1/2} = \sqrt{8ln(2)}\sigma = (2.35482)\sigma\ . \tag{19-2}$$

The Gaussian function maximizes to 1 at $t = t_o$ and is zero at $t = \pm\infty$. A plot of this function would be

Analog Gaussian Plot

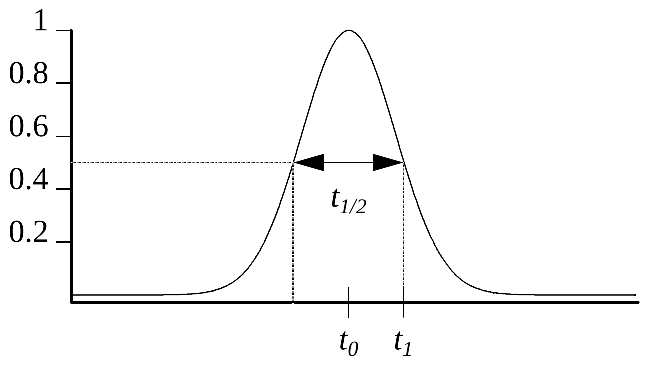


Figure 19-3 A Gaussian function depicting the peak maximum at $t = t_o$ and the linewidth at half-height. This function maximum is 1 and the end points tend to zero at $t = \pm\infty$. This plot was produced from the program Gplot.cc given at the end of this chapter.

The relationship between σ and $t_{1/2}$ is derived as follows.

$$\frac{f(t_1)}{f(t_o)} = \frac{0.5}{1} = \frac{exp[-(t_1-t_0)^2/(2\sigma^2)]}{exp[-(t_0-t_0)^2/(2\sigma^2)]}$$

$$\frac{1}{2} = exp\left[\frac{-(t_1 - t_0)^2}{2\sigma^2}\right] = exp\left[\frac{-(0.5t_{1/2})^2}{2\sigma^2}\right]$$

$$ln(0.5) = \frac{-(0.5t_{1/2})^2}{2\sigma^2} \qquad 2ln(2) = \frac{(0.5t_{1/2})^2}{\sigma^2} \qquad \frac{1}{2}t_{1/2} = \sigma\sqrt{2ln(2)}$$

$$t_{1/2} = \sqrt{8ln(2)}\sigma = (2.35482)\sigma \tag{19-3}$$

Because the Gaussian define above maximizes to a value of 1 whereas we can change its linewidth by altering the standard deviation, the integrated area under the curve varies with σ. If desired, a normalization factor may be placed in front of the Gaussian so that it's integrated intensity is 1 rather that its maximum height. This is done by multiplication by $1/N$ where N is given by

$$N = \int_0^\infty G(t)dt = \sqrt{2\pi}\sigma \tag{19-4}$$

a value that can be obtained as follows.

$$N^2 = \int_0^\infty exp\left[\frac{-(x - x_0)^2}{2\sigma^2}\right]dx\int_0^\infty exp\left[\frac{-(y - y_0)^2}{2\sigma^2}\right]dy = \int_0^\infty\int_0^{2\pi} exp\left[\frac{-r^2}{2\sigma^2}\right]rdrd\theta$$

$$= 2\pi\int_0^\infty exp\left[\frac{-r^2}{2\sigma^2}\right]rdr = 2\pi\sigma^2\int_0^\infty e^{-u}du = 2\pi\sigma^2$$

4.9.3 Discrete Mathematical Basis

A computer representation of any waveform, such as the Gaussian function, must be done using discrete mathematics. The waveform will be repersented by a specified number of points, N, and the previous analog function can be adjusted to evaluate at each point according to

$$G_i = e^{[-(i-i_0)^2/2\sigma^2]} \tag{19-5}$$

However, we will demand a few modifications of this formula to make it suitable for defining a pulse shape in NMR. Keep in mind that a pulse programmer in a spectrometer is limited to the same discrete mathematics. The applied pulse wave form is only a discrete approximation to the true pulse function. We will tailor our discrete formula according to the following two conditions.

1.) The Gaussian maximum will be centered in the middle of the Gaussian points. Since the waveform will be applied at a specified time in a pulse sequence, we can move the center of the pulse to any point in the sequence. For an array of N points, the center is given by[1]

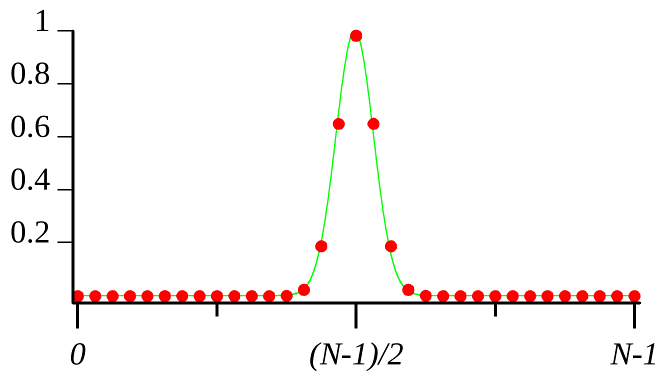
$$(N-1)/2 \tag{19-6}$$

and so our working equation becomes

$$G_i = e^{[-[i-(N-1)/2]^2/2\sigma^2]} = e^{[([2i-(N-1)]^2\ln(fact))/2\sigma^2]} \tag{19-7}$$

In this formula i is a non-negative integer, the point (or step) increment. The units on sigma must also be points but it need not be integer. Below is a plot of G_i using $N = 33$ and $\sigma = 1.1$ overlaid is an analog Gaussian.

Analog vs. Discrete Gaussian Plot



2.)Unlike the analog Gaussian, here we would like to set the value of σ, or equivalently the Gaus-

1. This if or C indexing, the first point indexed as 0 and the last point as N-1. Not that there may not actually be an evaluated point in the center. If N is even then the center will lie between two points of the discrete waveform.

sian spread, such that the intensity at the first point is a specified percentage of the maximum. Recall that a true Gaussian only approaches zero at an infinite distance away from the maximum. One would then need infinite time to attain a true Gaussian pulse shape, so we settle for shorter pulses by just truncating the Gaussian at some specified minimum height (~1%).

Rather than set the Gaussian width in terms of a σ value[1], a more appropriate choise would be to just specifiy the end-point intensity relative the the function maximum. In building Gaussian shaped pulses this is important because normally the initial Gaussian intensity is specified and should not be set to zero. We must choose a cutoff value which indicates the initial and final intensities of the discrete function based on a set percentage of the maximum intensity, $fact$, where $fact$ is the decimal form of a percent (e.g. 2% maximum is $fact = 0.02$). We can see how this will affect our discrete formula, or equivalently, what value of σ is required, by looking at either the first (i=0) or last (i=N-1) point.

$$fact = e^{[-[i-(N-1)/2]^2/2\sigma^2]}\Big|_{i=0} = e^{[-[i-(N-1)/2]^2/2\sigma^2]}\Big|_{i=N-1} \tag{19-8}$$

So that

$$fact = e^{-[(N-1)/2]^2/2\sigma^2} = e^{-(N-1)^2/8\sigma^2} \tag{19-9}$$

where $fact \in [0,1]$. Back solving this for the standard deviation produces

$$\sigma = (N-1)/\sqrt{-8\ln fact} \tag{19-10}$$

Putting this back into our original discrete Gaussian amplitude using

$$1/2\sigma^2 = -\ln(fact)/[(N-1)/2]^2 \tag{19-11}$$

The discrete Gaussian equation is then

$$G_i = e^{\ln(fact)[2i-(N-1)]^2/(N-1)^2} \tag{19-12}$$

How the discrete function behaves is shown in the next figure.

1. That would allow for either most points to be zero by selecting a very small standard deviation, or for having the defined Gaussian almost constant by selecting a very large standard deviation.

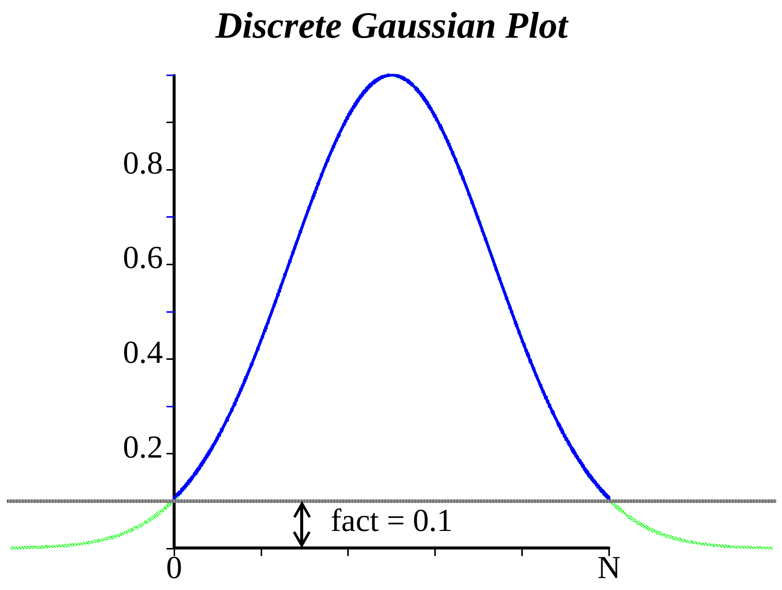


Figure 19-4 A discrete Gaussian function depicting the peak maximum at $(N-1)/2$. In this case the "linewidth" is set by the function intensity at the two endpoints, in decimal form percent of maximum peak height, fact.This plot was produced by program Gplot.cc given at the end of this chapter.

If we now check the Gaussian endpoints we find that

$$G_0 = G_{N-1} = fact$$

which is what we intended. Furthermore, the discrete Gaussian is symmetric, as can be proven by demonstrating that $G_{(N-1)-i} = G_i$.

$$G_{(N-1)-i} = exp\left[\frac{(2(N-1-i)-(N-1))^2\ln(fact)}{(N-1)^2}\right] = exp\left[\frac{(N-1-2i)^2\ln(fact)}{(N-1)^2}\right]$$

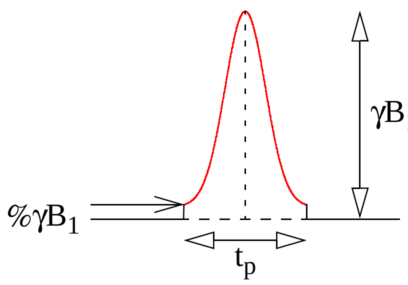
$$= exp\left[\frac{(2i-(N-1))^2\ln(fact)}{(N-1)^2}\right] = G_i$$

4.9.4 Discrete Pulse Mathematics

Having discussed the equations which apply to Gaussian functions, we turn our attention to construction of a Gaussian pulse. In this application the function merely defines the relative intensity of an applied rf-field in time. At each point in the discrete function, the rf intensity is adjusted to a new value and that value is maintained until the next point or the end of all points.

A Gaussian pulse is specified in part by a pulse length (t_p), a field strength at the maximum, (γB_1), and a percentage of this value that will be the rf intensity at the beginning of the pulse. This is depicted in the following figure.

Gaussian Pulse Parameters



Additionally, because instrument amplifiers cannot do an analog Gaussian intensity modulation, the pulse is broken up into a number of steps and this number also characterizes the pulse. We can readily depict this situation by drawing rectangles representing the rf-strengths during each point.

Gaussian Pulse Shape

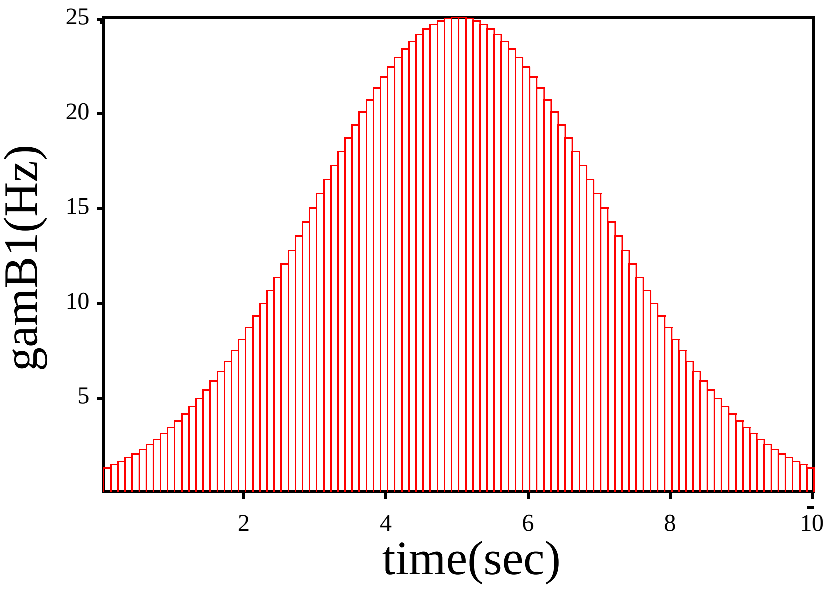
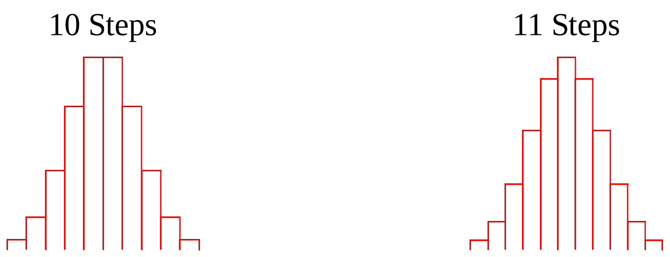


Figure 19-5 A Gaussian pulse wave form. The rf-field strength (gamB1) is discretely changed over a finite time increment based on the number of steps and the total length of the pulse. This plot was produced from the program Gplot.cc given at the end of this chapter.

Our plan is to maintain symmetric Gaussian waveforms, regardless of the number of point characterizing them. Examples are shown in the next figure.

Gaussian Pulse Symmetry



In practice there is a small delay between each step and each step will not be a true square wave as rf amplifiers cannot turn on and off instantaneously.

Now, the equation for the Gaussian intensity is given by

$$G_i = \gamma B_1 e^{\ln(fact)[2i-(N-1)]^2/(N-1)^2} \tag{19-13}$$

which is simply our previous formula multiplied by an rf-field strength γB_1. This strength is maintained for a specified time increment, Δt, where the total pulse length for the N steps is

$$t_p = N\Delta t \tag{19-14}$$

Note that the discrete Gaussian intensities, as written above, do not contain any time variables. Howevert the two are related because the "on-resonance" angle of rotation for any step is given by

$$\theta_i = G_i \times \Delta t \tag{20}$$

and the total "on-resonance" rotation due to the Gaussian pulse by

$$\theta_p = \sum_{i=0}^{N-1} G_{p,i} \times \Delta t = t_p \sum_{i=0}^{N-1} G_{p,i} \times \Delta t = \gamma B_1 t_p \sum_{i=0}^{N-1} G_i \tag{21}$$

Because the integral of the plain discrete Gaussian,

$$\sum_{i=0}^{N-1} G_i$$

depends on the number of steps taken, N, it is clear that the parameters

$$\theta_p \qquad t_p \qquad \gamma B_1 \qquad N$$

are related. Often the value of γB_1 is determined by setting the other three parameters according to

$$\theta_p \Big/ \left(t_p \sum_{i=0}^{N-1} G_i \right) = \gamma B_1 \tag{22}$$

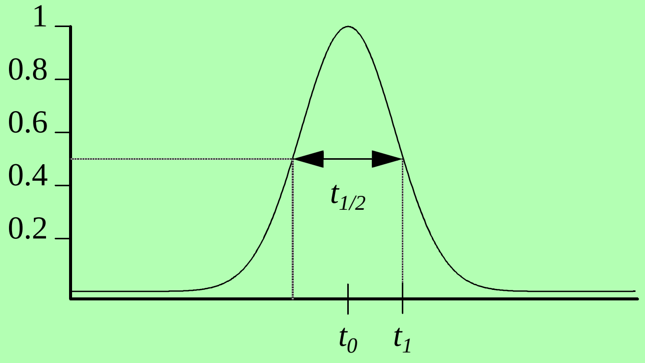
Gaussian Pulse Summary

Analog Gaussians

$$G(t) = exp\left[\frac{-(t-t_o)^2}{2\sigma^2}\right]$$

$$t_{1/2} = \sqrt{8ln(2)}\sigma$$

$$\int_0^\infty G(t)dt = \sqrt{2\pi}\sigma$$

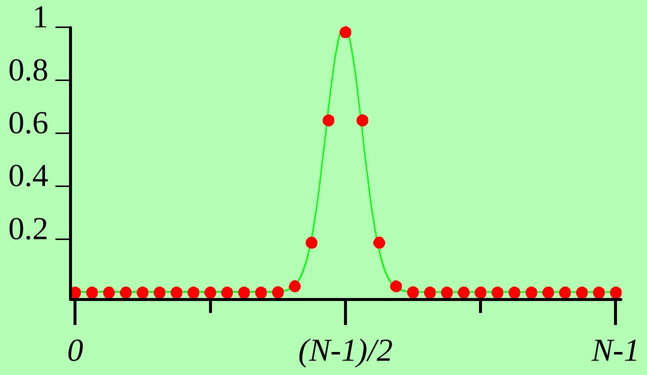


Discrete Gaussians

$$G_i = exp\left[\frac{-\left(i-\frac{1}{2}(N-1)\right)^2}{2\sigma^2}\right]$$

$$fact \in [0,1]$$

$$G_i\Big|_{\substack{max\ =\ 1\\ min\ =\ fact}} = e^{\left[\frac{(2i-(N-1))^2\ln(fact)}{(N-1)^2}\right]}$$

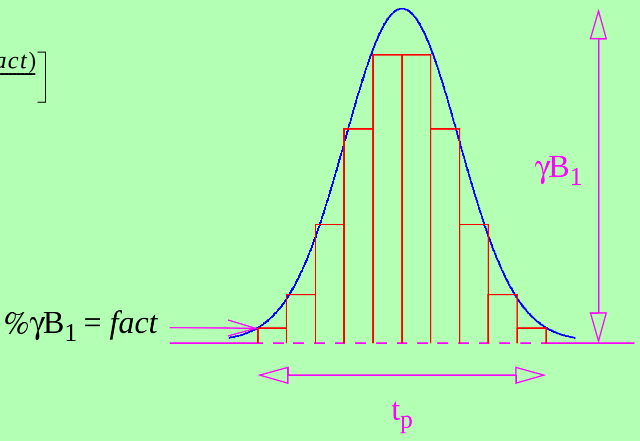


Discrete Gaussian Pulses

$$G_{p,i}\Big|_{\substack{max\ =\ \gamma B_1\\ min\ =\ fact}} = \gamma B_1 e^{\left[\frac{(2i-(N-1))^2\ln(fact)}{(N-1)^2}\right]}$$

$$t_p = N \times \Delta t$$

$$\gamma B_1 = \frac{\theta_p}{t_p \sum_{i=0}^{N-1} G_i}$$



4.9.5 Gaussian Pulses, No Relaxation

Without relaxation, each step i is given by the solution to the Liouville equation under a constant effective Hamiltonian in the rotating frame of the applied rf-field. In this case we can write

$$\hat{\underset{\sim}{\sigma}}_{i+1} = \hat{U}_i \hat{\underset{\sim}{\sigma}}_i U_i^{-1} \tag{22-1}$$

where $\hat{U}_i$ is a unitary propagator which evolves the system for time Δt under the Gaussian field strength G_i. The underscore tilde $\sim$ is meant to denote the rotating frame of the applied rf-field. Individual propagators are generated from the effective Hamiltonian

$$\hat{U}_i = \exp(-i\hat{H}_{i,\,eff}\Delta t) \tag{22-2}$$

where

$$\hat{H}_{i,\,eff} = \hat{H}_0 - \Omega_{rf}\hat{F}_{z,\,i} + G_i\hat{F}_{x,\,y} \tag{22-3}$$

Starting with the initial density operator, $\hat{\underset{\sim}{\sigma}}_0$, at the end of N steps we will have

$$\hat{\underset{\sim}{\sigma}}_N = \left(\prod_{i=0}^{N-1}\hat{U}_i\right)\hat{\underset{\sim}{\sigma}}_0\left(\prod_{i=0}^{N-1}\hat{U}_i\right)^{-1} \tag{22-4}$$

Note that since each the propagators evolves the density operator for a time Δt, at the end of the sequence the time will be the length of the applied shaped pulse

$$t_p = N\Delta t \tag{22-5}$$

We can define a Gaussian pulse propagator (no relaxation) to be

$$\hat{U}_{GP}(t_p, \gamma B_1, \Omega_{rf}, N) = \prod_{i=0}^{N-1}\hat{U}_i(\Delta t, \gamma B_1, \Omega_{rf}) = \hat{U}_{N-1}\ldots\hat{U}_1\hat{U}_0 \tag{22-6}$$

Evolution under a Gaussian pulse is then given by

$$\hat{\underset{\sim}{\sigma}}(t_p + t_0) = \hat{U}_{GP}(t_p, \gamma B_1, \Omega_{rf}, N)\hat{\underset{\sim}{\sigma}}(t_0)\hat{U}_{GP}^{-1}(t_p, \gamma B_1, \Omega_{rf}, N) \tag{22-7}$$

4.9.6 Gaussian Pulses, With Relaxation

We shall now repeat the mathematical flow of the previous sections but account for relaxation in a rigorous fashion using WBR theory. In this case spin system evolution is given by

$$\underset{\sim}{\sigma}_{i+1} = \exp(-i\hat{\hat{L}}_i\Delta t)\Delta\underset{\sim}{\sigma}_i + \underset{\sim}{\sigma}_{i,ss} \tag{22-8}$$

where $\hat{\hat{L}}_i$ is the Liouvillian superoperator which dictates spin system evolutionn, $\Delta\hat{\hat{\underset{\sim}{\sigma}}}_i$ the difference density operator at point i, and $\hat{\hat{\underset{\sim}{\sigma}}}_{i,ss}$ the steady state at that same point. The difference density operator is given by

$$\Delta\hat{\hat{\underset{\sim}{\sigma}}}_i = \hat{\hat{\underset{\sim}{\sigma}}}_i - \hat{\hat{\underset{\sim}{\sigma}}}_{i,ss} \tag{22-9}$$

wheras the steady-state matrix itself is determined from

$$\hat{\underset{\sim}{\sigma}}_{i,ss} = \frac{\hat{\hat{\underset{\sim}{R}}}_i}{\hat{\underset{\sim}{L}}_i}\hat{\sigma}_{eq} \tag{22-10}$$

the superoperator $\hat{\hat{\underset{\sim}{R}}}_i$ containing all Liouvillian terms except those from the commutation Hamiltonian superoprator.

Because we plan to cycle through many steps in the application of our Gaussian pulse, it is convenient to rewrite equation (22-8) interms of superoperator propagators.

$$\underset{\sim}{\sigma}_{i+1} = \Gamma_i\underset{\sim}{\sigma}_i + \underset{\sim}{\sigma}_{i,ss} \tag{22-11}$$

4.9.7 Gaussian Pulse Equations

In this section we regroup the applicable equations regarding ***Gaussian Pulses***

Gaussian Shaped Pulse Equations

Overall

$$\langle Op(t)\rangle = Tr\{Op\cdot\sigma(t)\} = \langle Op^{\dagger}|\sigma(t)\rangle =$$

Unitary Transformation, Hilbert Space

$$\langle Op(t_k)\rangle = \sum_p A_p[B_p]^k = Tr\left\{Op\cdot \boldsymbol{U}^k\sigma(t_o)[\boldsymbol{U}^{-1}]^k\right\}$$

$$A_{\alpha\alpha'} = \langle\alpha|Op|\alpha'\rangle\langle\alpha'|\sigma(t_o)|\alpha\rangle$$

$$B_{\alpha\alpha'} = \langle\alpha'|\boldsymbol{U}|\alpha'\rangle\langle\alpha|[\boldsymbol{U}^{-1}]|\alpha\rangle$$

$$p = \alpha\alpha' \quad \forall \quad \langle\alpha|Op|\alpha'\rangle \neq 0$$

Expectation Value at Time t_k

$$\sigma(t_k) = \boldsymbol{U}^k\sigma(t_o)[\boldsymbol{U}^{-1}]^k$$

$$\boldsymbol{U} = e^{-i\boldsymbol{H}(\Delta t)}$$

$$\langle Op(t_k)\rangle = \sum_p A_p[B_p]^k$$

Non-Unitary Transformation, Liouville Space

$$\langle Op(t_k)\rangle = \sum_p A_p[B_p]^k = Tr\{Op\cdot\Gamma^k_{RP}\sigma(t_o)\}$$

$$A_{\alpha\alpha'} = \langle 1|Op^{\dagger}S|\alpha\alpha'\rangle\langle\alpha\alpha'|S^{-1}\sigma(t_o)|1\rangle$$

$$B_{\alpha\alpha'} = \langle\alpha\alpha'|\Lambda|\alpha\alpha'\rangle$$

$$\sigma(t_k) = \Gamma^k\sigma(t_o)$$

$$p = \alpha\alpha' \quad \forall \quad \langle 1|Op^{\dagger}S|\alpha\alpha'\rangle \neq 0$$

Redfield Theory, Liouville Space

$$\langle Op(t_k)\rangle = \sum_p A_p[B_p]^k + Tr\{Op\cdot\hat{\sigma}_{inf}\} = Tr\{Op\cdot[\Gamma^k(\sigma(t_o)-\hat{\sigma}_{inf})+\hat{\sigma}_{inf}]\}$$

$$A_{\alpha\alpha'} = \langle 1|Op^{\dagger}S|\alpha\alpha'\rangle\langle\alpha\alpha'|S^{-1}[\sigma(t_o)-\hat{\sigma}_{inf}]|1\rangle$$

$$B_{\alpha\alpha'} = \langle\alpha\alpha'|\Lambda|\alpha\alpha'\rangle$$

$$\sigma(t_k) = \Gamma^k\{(t_o)-\hat{\sigma}_{inf}\}+\hat{\sigma}_{inf}$$

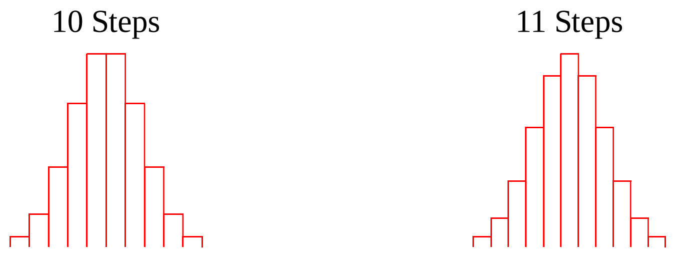
$$\Gamma = e^{-L\Delta t}$$

$$p = \alpha\alpha' \quad \forall \quad \langle 1|Op^{\dagger}S|\alpha\alpha'\rangle \neq 0$$

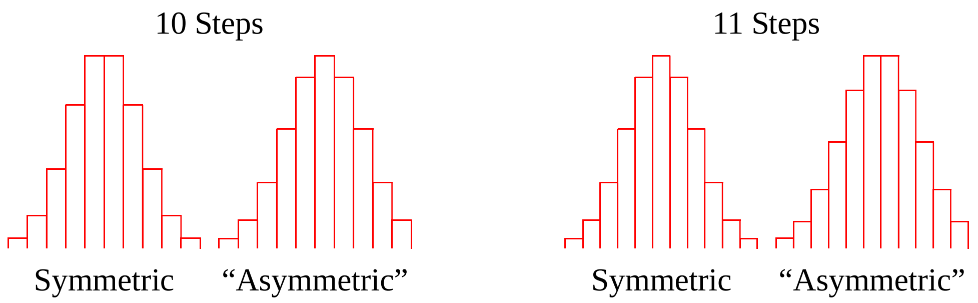
4.9.8 Final Notes

A discrete Gaussian function, that is to say a finite array of points with values related to a Gaussian distribution, will NOT be zero at its endpoints. Rather it will be some finite value the may become close to zero within the machine precision. In building Gaussian shaped pulses this is important because normally the initial Gaussian intensity is specified (say at 5% maximum) and should not be set to zero.

(23)



Additionally, because instrument amplifiers cannot do an analog Gaussian intensity modulation, the pulse is broken up into a number of steps and this number also characterizes the pulse. Examples are shown in the next figure. For a given number of steps, the Gaussian can be broken up symmetrically or “asymmetrically”. The former requires more sophisticated tracking of the individual step intensities but will in principle produce better excitation profiles. The latter is mathematically easier to generate and likely to be what is supplied with a spectrometer.



Typically one will take a number of steps ~10^3 so the differences between these two constructs becomes small. In practice there is a small delay between each step and the each step is not a true square wave as the amplifier does not turn on and off instantaneously.

4.10 Gaussian Pulse Parameters

4.10.1 Introduction

Gaussian pulses are often used in NMR as they can be tailored to be highly selective (i.e. covering a selected frequency range) with relatively small amplitude and phase distortions. In GAMMA, such pulses are treated as a special cases (rather than simply a general shaped pulse[1]) because the pulse shape symmetry allows for significant computational savings. The module ***P_Gaussian*** provides a variety of functions pertaining to Gaussian shaped pulses. Of interest here are the functions in ***P_Gaussian*** that either return propagators which evolve the density operator or act on the density operator directly.

4.10.2 Gaussian Pulse Parameters

A Gaussian pulse is specified in part by a pulse length (t_p), a field strength at the maximum, (γB_1), and a percentage of this value that will be the rf intensity at the beginning of the pulse ($\%\gamma B_1$). Additionally, because instrument amplifiers cannot do an analog Gaussian intensity modulation, the pulse is broken up into a number of steps (N). We can readily depict this situation in the following figure, using rectangles to represent the rf-strengths during each point.

Four Gaussian Pulse Parameters

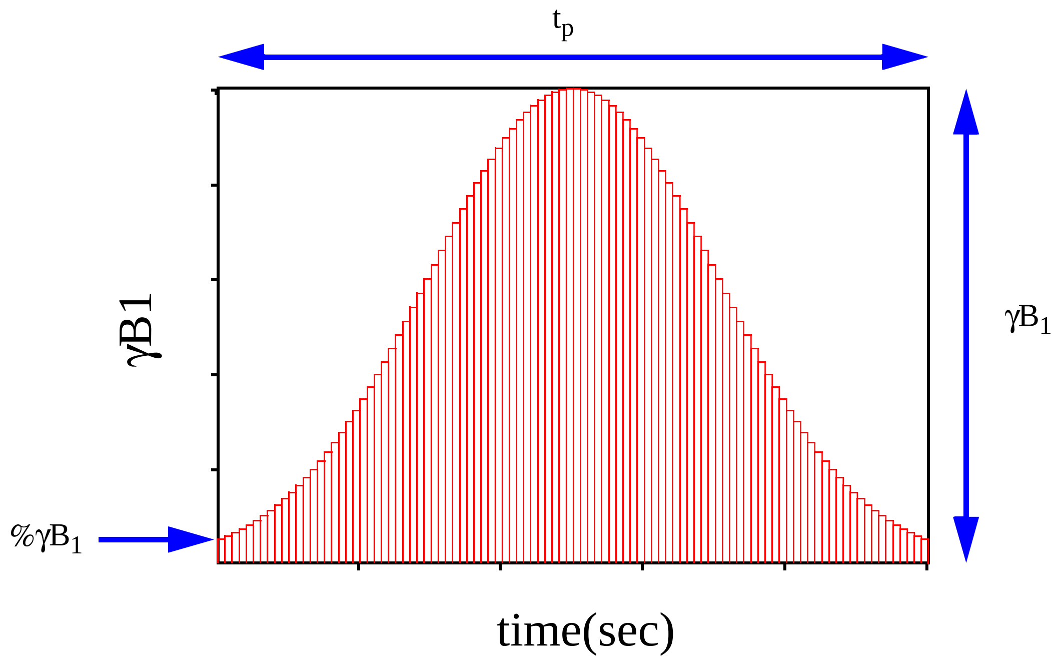


Figure 19-6 A Gaussian pulse wave form. The rf-field strength (gamB1) is discretely changed over

1. GAMMA users may construct arbitrary shaped pulses by simply supplying a vector containing the desired waveform and a few other pulse parameters. Look at the documentation regarding shaped pulses to see how that is accomplished. We recommend that you use the functions in the Gaussian pulse module when your programs require such pulses. They are easier to use and the routines faster computationally.

a finite time increment based on the number of steps and the total length of the pulse. The entire pulse waveform is determined from four parameters $\{t_p, \gamma B_1, N, \%\gamma B_1\}$. The program which produced this plot can be found in the documentation for the ***P_Gaussian*** module.

Three other parameters are required for a Gaussian pulse, the pulse offset (or carrier frequency), pulse phase, and the pulse channel. That makes a total of seven parameters for complete characterization of a Gaussian pulse in GAMMA. Indeed, use of GAMMA's Gaussian pulses is quite simple if the user has a clear understanding of the means by which the seven parameters are set in a the program. This will be the topic of the next sections.

4.10.3 Defining a Gaussian Pulse Directly

This task is accomplished by specifying the seven parameters that define a Gaussian pulse. For example, the following code will do (we will show how to make and apply the pulse later):

4.10.4 Defining a Gaussian Interactively

As in the last section, we need to specify the seven parameters that define a Gaussian pulse. In this case we need to have the program itself ask for the parameters as the program is run.

4.10.5 Defining a Gaussian Pulse in an External File

GAMMA provides a very simple means of defining a Gaussian pulse in an external "parameter" file. The parameter file is simply an ASCII file which contains parameters that a GAMMA Gaussian pulse type will recognize. A GAMMA parameter is a line in a file having the format

Name (type) : value - optional comment

There are 9 parameter names that will be recognized as defining a Gaussian pulse.

Table 1: Gaussian Pulse Parameters

Parameter Keyword	Assumed Units	Examples Parameter (Type) : Value - Statement
Gstps	none	Gstps (0) : 41 - Steps in Gaussian Pulse
Gang	degrees	Gang (1) : 90.0 - Gaussian Pulse Angle
Glen	seconds	Glen (1) : 0.1 - Gaussian Pulse length
Gcut	none	Gcut (1) : 0.025 - RF cutoff level (%GgamB1 -> 2.5%)
GW	Hz	GW (1) : 400.0 - Gaussian Pulse Carrier Frequency
Giso	none	Giso (2) : 1H - Gaussian Pulse Channel
GgamB1	Hz	GgamB1 (1) : 55. - Gaussian Pulse RF Field Strength

Table 1: Gaussian Pulse Parameters

Parameter Keyword	Assumed Units	Examples Parameter (Type) : Value - Statement
Gphi	degrees	Gphi (1) : 0.0 - Gaussian Pulse Phase
Gspin	none	Gspin (0) : 0 - Gaussian Spin Selectivity

Of course, there are only seven parameters necessary to completely characterize the pulse. Two of the above parameters are redundant. The first redundancy comes from the three parameters {Gang, Gtime, GgamB1}. Only two of the three need to be specified. The set {Gang, Gtime} will be used preferentially if all three are set in the parameter file. The second redundancy comes in the selectivity. Either {Gspin} or {Gwrf, Giso} Sets Selectivity. If Gspin is set it will be preferentially used and override any Giso and Gwrf settings. However, note that use of Gspin DOES NOT set the Gaussian pulse selectivity to only affect a particular spin (which is not experimentally possible for strongly coupled or overlapping spins). Rather, it sets the Gaussian pulse carrier to be at the spin Larmor frequency.

Two other points are worth mentioning. If one has a homonuclear spin system the selectivity does not have to be set. Thus, neither Giso nor Gspin need be set if the there is only 1 channel and the pulse does not need to be spin selective. Some simulations utilize multiple Gaussian pulses, so there is often a need to define more than one pulse in a parameter file. This may be accomplished by adding on a (#) to the Gaussian parameter name where # is simply an integer which is a pulse index. The set of parameters which define a pulse are then all set with the same number in their names and the number used in the GAMMA program which reads the parameters.

Note that pulse parameters can be mixed with other parameters in a single ASCII file. For example, you can readily include the Gaussian pulse definition in the same file that defines your spin system. The code and file would look something like

4.10.6 Constructing a Gaussian Pulse Propagator

d look something like

4.10.7 Example: Gaussian Pulse Profile

This example takes a single spin and applies a specified Gaussian pulse (90y). It repeats this process while moving the spins chemical shift through the frequency range over which the profile is desired. The response versus pulse offset for the spin is plotted, both with and without (magnitude) the phase information present.

Single Spin Gaussian Pulse Profile

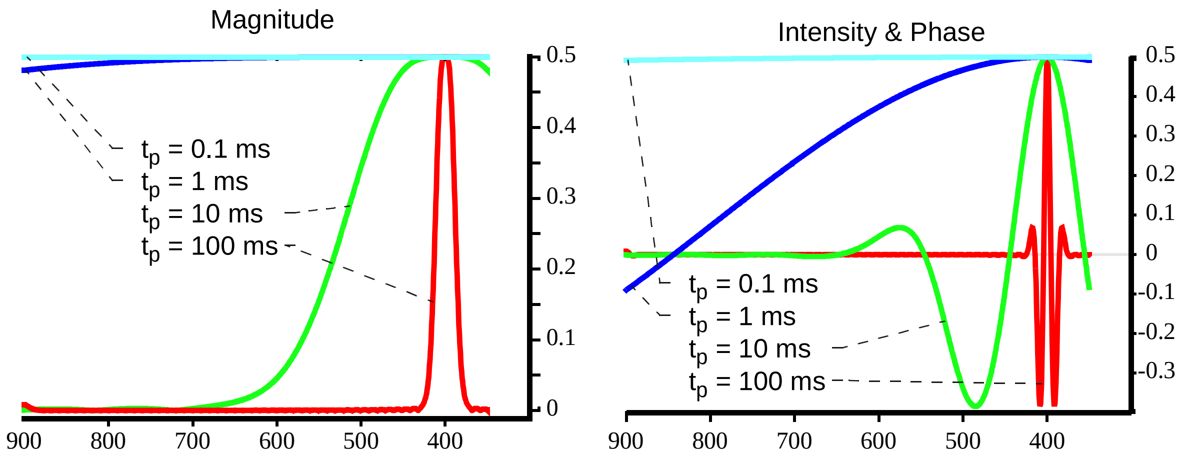


Figure 19-7 The plots were produced from successive runs of the program Gprofile2.cc on page 71. In all instances the Gaussian was applied at an offset of 400 Hz and the generated profiles constructed between 350 and 900 Hz with a block size of 1024 points. The program automatically sets the Gaussian to be a 90y pulse of 51 steps and an endpoint cutoff of 2.5% maximum intensity. Of minor interest is the small excitation produces near 900 Hz in the more selective Gaussian's. This is a consequence of using only 51 steps for the pulse shape and such effects disappear a the number of steps increase.

This simulation follows the general rule of thumb in NMR: short strong (hard) pulses promote even excitation over a broad frequency range and weak long (soft) pulses are selective in that they excite over a narrow frequency range. Notice that a second rule of thumb is also followed: Long pulses induce phase distortions off resonance. This is due primarily to the “dephasing” of the transitions during the time it takes to get the magnetization down into the xy-plane. At the end of a long 90 pulse, not all magnetization vectors are aligned along an axis perpendicular to the pulse because they have undergone precession during the pulse itself. In an ideal situation, the resulting spectrum would be phase adjusted using a 1st order phase correction. However, that assumes a linear response to the pulse which is not strictly the case - especially for strongly coupled spin systems and non-uniform pulse waveforms. So, phase distortions in a spectrum due to a long pulse can be difficult to remove by simple phase corrections.

4.10.8 Example: Gaussian 90 Pulse

This example takes a spin system and applies a specified Gaussian pulse (90y). It reads in both the spin system and the Gaussian pulse definition from a single GAMMA parameter file.

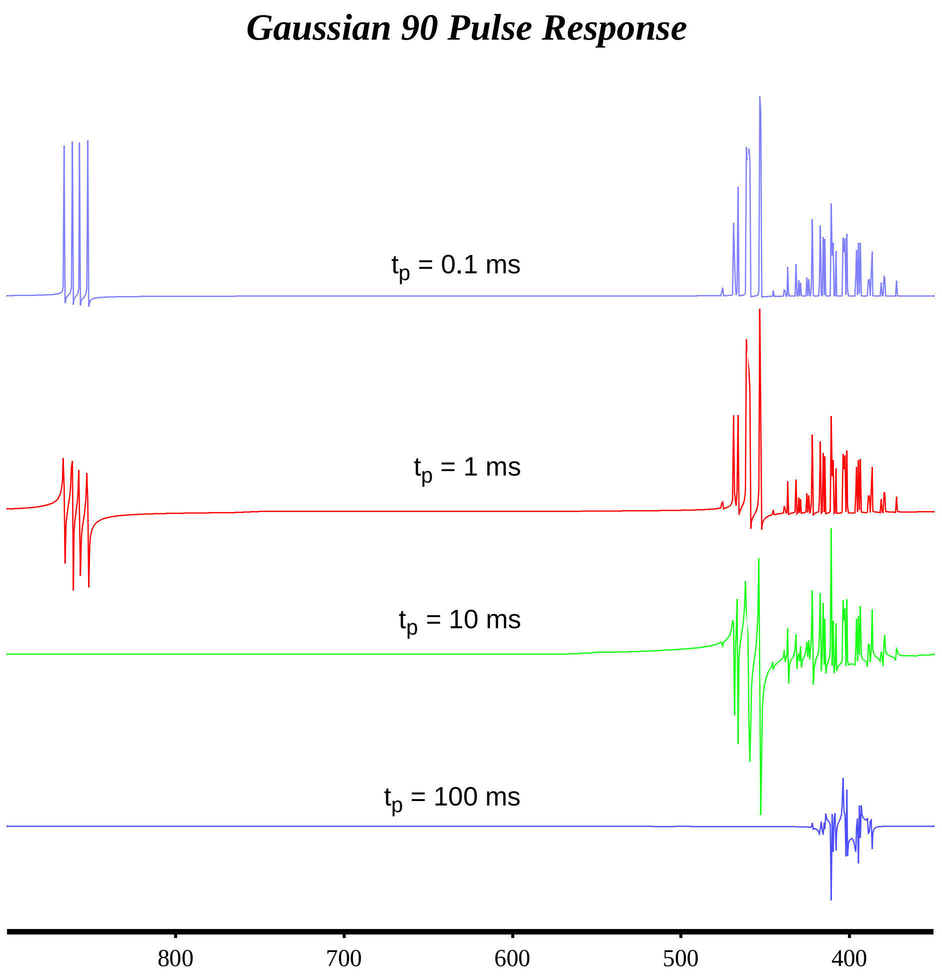


Figure 19-8 The plots were produced from successive runs of the program Gpulse0.cc on page 72. In all instances the program was given the parameter file GlutamicA.sys on page 73 which contains both a spin system definition and the Gaussian pulse parameters. Only the Gaussian pulse length was changed between the successive runs. The executable (a.out) was repeatedly invoked with the command “a.out GlutamicA.sys 350 900 1024 0.02” which set the plot range to span 350-900 Hz, the block size to 1K and the single quantum transition linewidths to .02 Hz.

Note that, although there is good selectivity with the 100 ms pulse, the inherent phases are terrible. In the next examples we shall attempt to minimize the phase errors by two different means. First, rather than using a 90 pulse we can attempt to use a Gaussian 270 pulse which has some self-refocusing properties which can reduce such problems[1]. Second we can attempt to perform a phase correction, either 1st order or using a single spin’s pulse response to the pulse.

1. Or worsen them in strongly coupled spin systems!

4.10.9 Example: Gaussian 270 Pulse

This example takes a spin system and applies a specified Gaussian pulse (270y). It reads in both the spin system and the Gaussian pulse definition from a single GAMMA parameter file.

Gaussian 270 Pulse Response

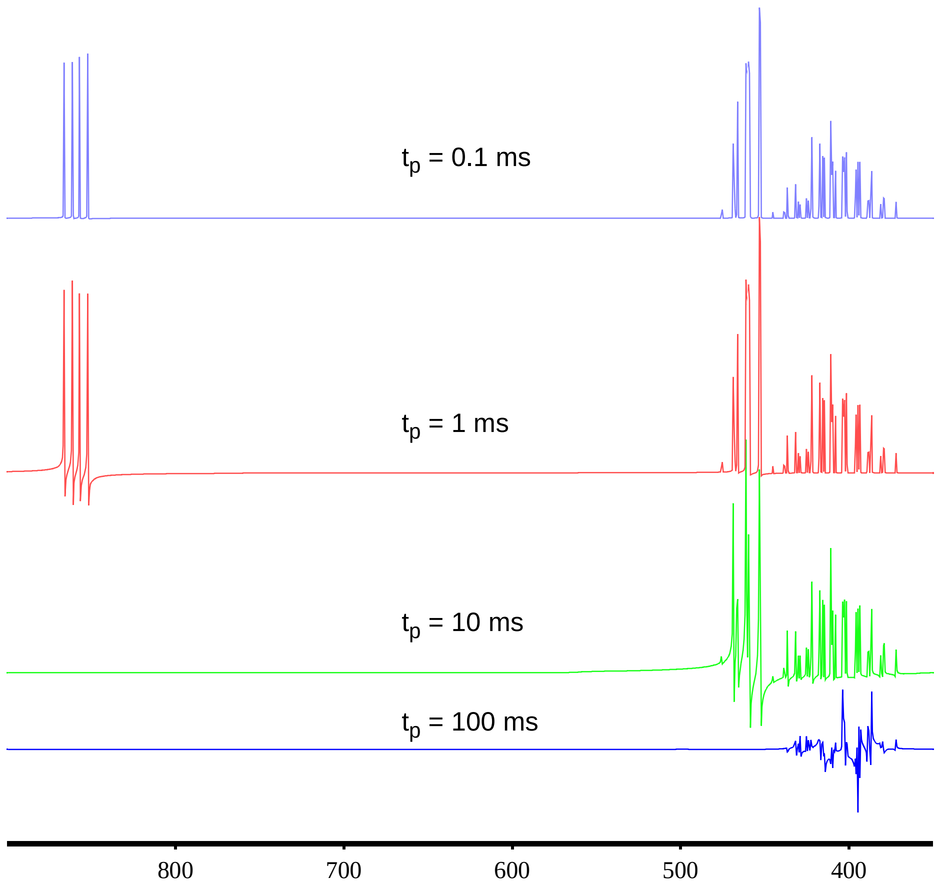


Figure 19-9 The plots were produced from successive runs of the program Gpulse1.cc on page 74. In all instances the program was given the parameter file GlutamicA2.sys on page 73 which contains both a spin system definition and the Gaussian pulse parameters. Only the Gaussian pulse length was changed between the successive runs. The executable (a.out) was repeatedly invoked with the command “a.out GlutamicA2.sys 350 900 1024 0.02” which set the plot range to span 350-900 Hz, the block size to 1K and the single quantum transition linewidths to 0.02 Hz.

In comparison with the previous simulation which used a 90 Gaussian pulse we see that there is some improvement in the phase behavior. However, there still remains quite a bit of dispersive nature to the multiplet with a 100 ms pulse.

4.11 Example: Gaussian Pulse, Profile Corrected

This example takes a spin system and applies a specified Gaussian pulse. It reads in both the spin system and the Gaussian pulse definition from a single GAMMA parameter file. In this case, it also creates a single spin response profile which displays how magnetization if affected by offset. Additionally, it adjusts the phases based on the single spin response - a rather robust phase correction.

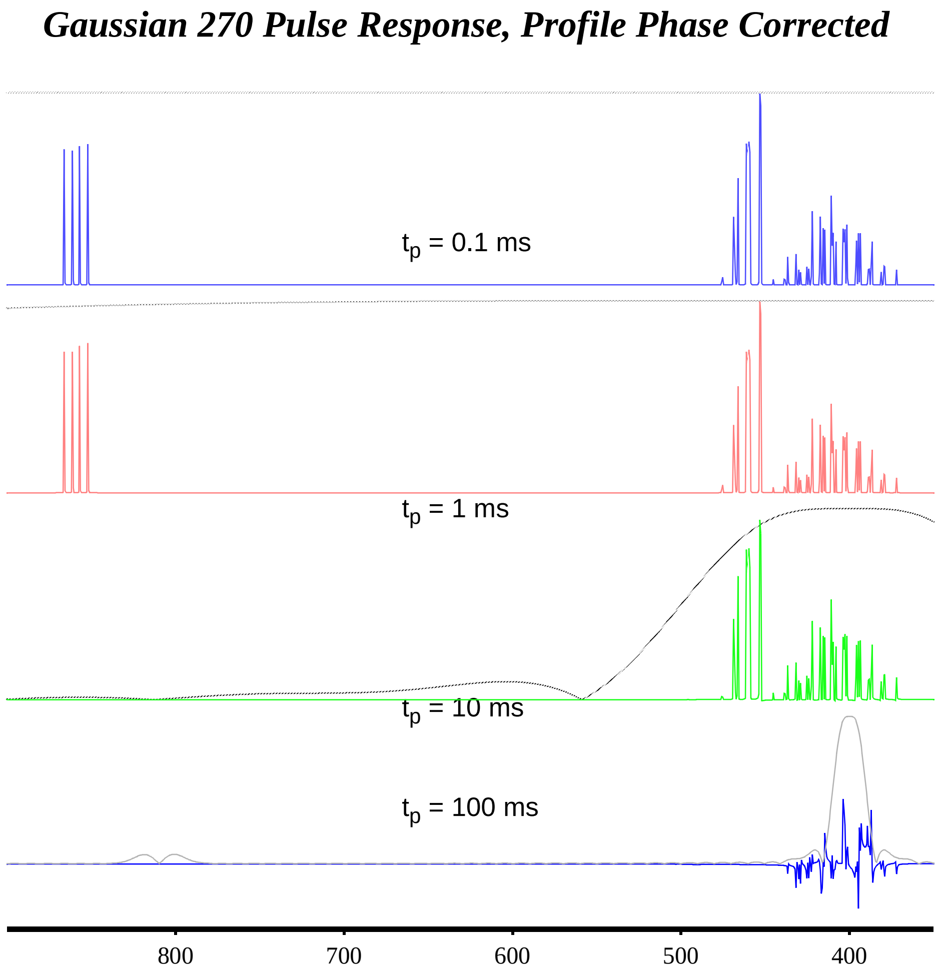


Figure 19-10 Plots produced from successive runs of Gpulcorr2.cc on page 75. The program was given the parameter file GlutamicA2.sys on page 73 which contains both a spin system definition and the Gaussian pulse parameters. Only the Gaussian pulse length was changed between the successive runs. The executable (a.out) was repeatedly invoked with the command “a.out GlutamicA2.sys 350 900 1024 0.02” which set the plot range to span 350-900 Hz, the block size to 1K and the single quantum transition linewidths to 0.02 Hz. The single spin profiles (magnitudes) are shown above the spectra in each case. (Note - a 90 Gaussian phase corrects better here!)

A quick comparison with the previous two simulations indicates that use of the single spin profile produces superior spectra. Unfortunately, this method would be time consuming and difficult to do experimentally. The 100 ms correction suffers from too few steps (41) and too high of end intensity (2.5%) characterizing the Gaussian, evident from the intensity near 800 Hz.

4.12 Example: Gaussian Pulse, Linear Phase Correction

Since we only have zero and first order phase corrections at our disposal on a spectrometer (without some fudging), it is perhaps worthwhile to examine our ability to use such a correction applied to the simulated spectra using selective Gaussians. Noting how difficult phase correction was in the previous example when the Gaussian was very long (selective), we expect linear phase correction to not perform very well.

Gaussian 90 Pulse Response, Standard Phase Correction

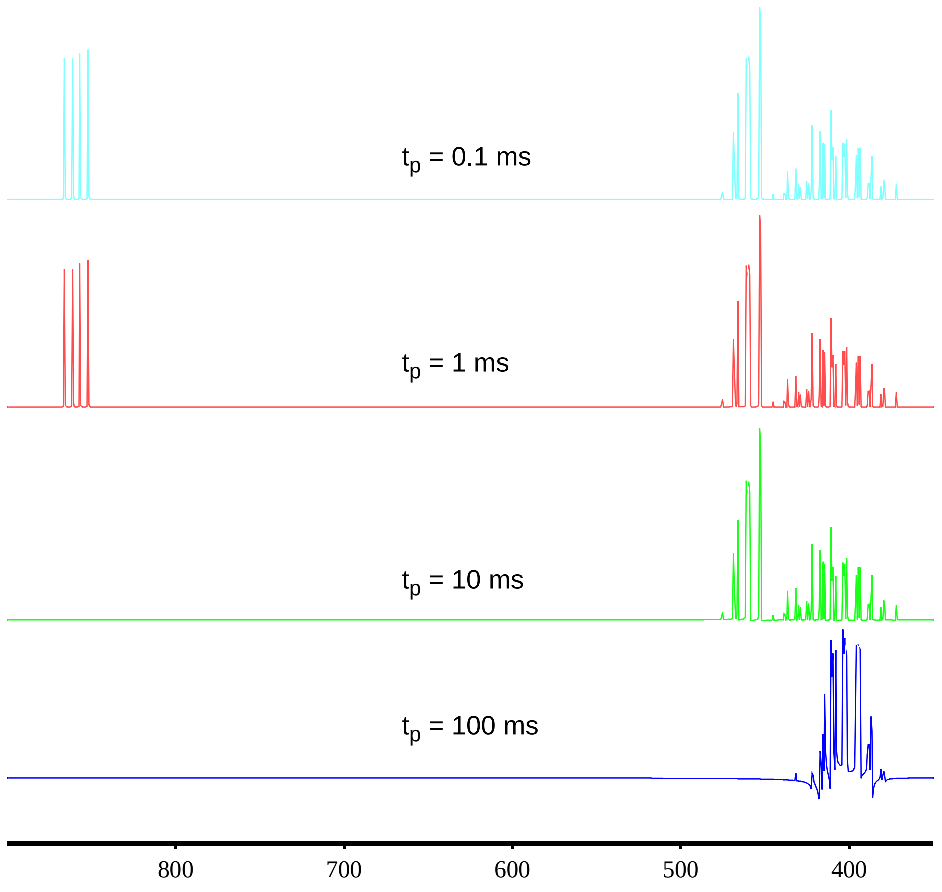


Figure 19-11 Plots produced from successive runs of Gpulpcorr2.cc on page 77. The program was given the parameter file slightly adjusted from GlutamicA2.sys on page 73 which contains both a spin system definition and the Gaussian pulse parameters. The Gaussian pulse length was changed between the successive runs. The pulse angle and phase were both set to 90. The executable (a.out) was repeatedly invoked with the command "a.out GlutamicA2.sys 350 900 1024 0.02" which set the plot range to span 350-900 Hz, the block size to 1K and the single quantum transition linewidths to 0.02 Hz.

Surprisingly, this type of phase correction works quite well. Again, I'll point out that one cannot directly compare the 100 ms run here with the previous calculation as the 90 pulse in general seems to phase correct better than a 270 pulse in strongly coupled systems.

Gprofile2.cc

```
/* Gprofile2.cc ****************************************************************-*-c++-*-
**                                                                                      **
**  This program plots a Gaussian shaped pulse profile.  The response                  **
**  of a single spin to the shaped pulse is measured versus rf-offset.                 **
**  This version extends Gprofile1.cc by allowing for the plot to be                   **
**  asymmetric about 0 Hz.  Thuse the pulse can be applied at any                      **
**  frequency and the plot can between any two frequencies.                            **
**                                                                                      **
**  Author:      S.A. Smith                                                             **
**  Date:        July 3 1996                                                            **
**  Last Update: July 3 1996                                                            **
**                                                                                      **
******************************************************************************************/

#include <gamma.h>                                      // Include all of GAMMA

main(int argc, char* argv[])
  {
  cout << "\n\n\t\t\t   GAMMA 1D NMR Simulation Program";
  cout << "\n\t\t\tGaussian Pulse Profile, No Relaxation\n\n";
//                  Set Gaussian Pulse Parameters

// (Set As A 90y Pulse Of 51 Steps, No Offset, No Phase, 2.5% Cutoff)
  int qn = 1;                                           // Query value
  Gpuldat Gdata;                                        // For Gaussian pulse params
  double tp;                                            // Gaussian pulse length
  query_parameter(argc, argv, qn++,                     // Read in this value
      "\n\tGaussian Pulse Length (sec)? ", tp);
  double Wrf;                                           // Gaussian pulse offset
  query_parameter(argc, argv, qn++,                     // Read in this value
      "\n\tGaussian Pulse Offset(Hz)? ", Wrf);
  Gdata.N = 51;                                         // Set pulse steps
  Gdata.Wrf = Wrf;                                      // Set pulse offset
  Gdata.Iso = String("1H");                             // Set pulse selectivity
  Gdata.tau = tp;                                       // Set the pulse length
  Gdata.fact = 0.025;                                   // Set pulse cutoff (2.5%)
  Gdata.phi = 0.0;                                      // Set pulse phase
  Gdata.gamB1 = GgamB1(90.0, tp, 51, 0.025);            // Set the pulse strength
  print_Gpulse(cout, Gdata);                            // Print Gaussian pulse params
//                         Set the Profile Parameters
  int npts;                                             // Number of points
  query_parameter(argc, argv, qn++,                     // Read in this value
           "\n\tProfile Block Size? ", npts);
  double Flow, Fhigh;                                   // Profile frequency limits
  query_parameter(argc, argv, qn++,                     // Read in this value
    "\n\tProfile Plotted Low Frequency (Hz)? ", Flow);
  query_parameter(argc, argv, qn++,                     // Read in this value
    "\n\tProfile Plotted High Frequency (Hz)? ", Fhigh);
  double delW = (Fhigh-Flow)/double(npts -1);           // Frequency increment
//                      Set the Unchanging Entities
  spin_system sys(1);                                   // A single proton
  gen_op D = Fm(sys);                                   // Detect F-
  gen_op sigma0 = sigma_eq(sys);                        // System at equilibrium
  gen_op Fxy = Fy(sys);                                 // RF field Ham. (rot. fr.)
  gen_op wFz = complex(Gdata.Wrf)*Fz(sys);              // RF Offset (rot. fr.)
//                        Set the Changing Entities
  matrix mx;                                            // Matrix for transitions
  gen_op Heff;                                          // Effective Hams
  gen_op sigmap;                                        // Prepared density oper
  gen_op UGauss;
  gen_op H;
  complex z;
//                        Perform The Simulation
  row_vector profile(npts), profnorm(npts);             // This for profile & magnitude
  double offset = Flow;
  for(int i=0; i<npts; i++)
    {
    sys.shift(0, offset);                               // Shift relative to pulse
    H = Ho(sys) + wFz;                                  // H in rotating frame
    UGauss = Gpulse_U(H, Fxy, Gdata);                   // Gaussian pulse propagator
    sigmap = evolve(sigma0, UGauss);                    // Evolve under the pulse
    z = trace(D, sigmap);                               // Get xy-magnetization
    profile.put(z,i);                                   // Store magnetization at i
    profnorm.put(norm(z),i);                            // Store magnetization at i
    offset += delW;                                     // Increment the offset
    }

  cout << "\n\n";                                       // Keep screen nice
  cout.flush();                                         // Print all before gnuplot
  GP_1D("profile.gnu", profile,0,Flow,Fhigh);           // Spectrum out in gnuplot
  GP_1D("profnorm.gnu", profnorm,0,Flow,Fhigh);         // Magnitudes out in gnuplot
  ofstream gnuload("gnu.dat");                          // File of gnuplot commands
  gnuload << "set data style line\n";                   // Set 1D plots to use lines
  gnuload << "plot \"profile.gnu\"";                    // Plot the magnetization response
  gnuload << ", \"profnorm.gnu\"\n";                    // Plot the magnitization magnitude
  gnuload << "pause -1 \'<Return> To Exit \n";          // Pause before quitting gnuplot
  gnuload << "exit\n";                                  // Exit gnuplot
  gnuload.close();                                      // Close gnuplot command file
  system("gnuplot \"gnu.dat\"\n");                      // This actually does plot to screen

  cout << "\n\n";                                       // Keep the screen nice
  }
```

Gpulse0.cc

```
/* Gpulse0.cc *****************************************************************-*-c++-*-
**                                                                                      **
**        NMR 1D Simulator Using A Gaussian Shaped Pulse                                **
**                                                                                      **
**  This program is an automated 1D NMR spectral simulator using                       **
**  shaped Gaussian pulses.  It runs inteactively, asking the user to                  **
**  supply a parameter file filename as well as plot parameters.  The                  **
**  system input is simply pulsed by the specified Gaussian.                           **
**                                                                                      **
**  This program does not include relaxation effects.  Also, the plot                  **
**  is immediate in gnuplot, so this will die if gnuplot is that                       **
**  program is not accessible. Finally, I made the y-axis the default                  **
**  axis for the pulse (I don't know why anymore....) so that a 0 phase                **
**  for input results in absorption on resonance using F- to detect, if                **
**  the pulse is 90 that is.                                                           **
**                                                                                      **
** Author:    S.A. Smith                                                                **
** Date:      7/2/96                                                                    **
** Update:    7/2/96                                                                    **
** Version:   3.5                                                                       **
**                                                                                      **
******************************************************************************************/

#include <gamma.h>                                      // Include GAMMA

main (int argc, char* argv[])

  {
  cout << "\n\n\t\t\tGAMMA 1D NMR Simulation Program";
  cout << "\n\t\t\tGaussian Pulses, No Relaxation\n\n";
//                      Read in System and Pulse Parameters

  int qn = 1;
  String filein;                                         // Input system file name
  query_parameter(argc, argv, qn++,                     // Get system file name
       "\n\tInput Parameter File? ", filein);
  sys_dynamic sys;                                      // A spin system
  sys.read(filein);                                     // Read in the system
  Gpuldat Gdata = read_Gpulse(filein, sys);             // Read Gaussian pulse params
  print_Gpulse(cout, Gdata);                            // Print Gaussian pusle params
  sys.offsetShifts(Gdata.Wrf);                          // Center system at pulse
//                  Determine Isotope Detection Type, Set Variables

 String IsoD;
 query_isotope(sys, IsoD);                              // Get the detection isotope
 gen_op sigma = sigma_eq(sys);                          // Set density matrix equilibrium
 gen_op H = Ho(sys);                                    // Isotropic Hamiltonian
 gen_op detect = Fm(sys, IsoD);                         // Set detection operator to F-
 acquire1D ACQ(detect, H, 1.e-3);                       // No relaxation during acquisition (Ho)
  gen_op Fxy = Fy(sys, Gdata.Iso);                      // For Gaussian RF Hamiltonian
  gen_op UGauss = Gpulse_U(H,Fxy,Gdata);                // Gaussian pulse propagator
//                         Apply The Pulse Sequence

  sigma = evolve(sigma, UGauss);                        // Apply Gaussian pulse
  matrix mx = ACQ.table(sigma);                         // Set up 1D acquisition
//                         Set The Plotting Parameters

  double Fstart, Fend;                                  // Number of points in FID
  query_parameter(argc, argv,                           // Get number of points
    qn++, "\n\tPlot Starting Frequency? ", Fstart);
  query_parameter(argc, argv,                           // Get number of points
    qn++, "\n\tPlot Final Frequency? ", Fend);
  int N;
  query_parameter(argc, argv,                           // Get number of points
    qn++, "\n\tPlot Points? ", N);
  double lwhh;                                          // Half-height linewidth
  query_parameter(argc, argv,                           // Get number of points
    qn++, "\n\tPlot Linewidths? ", lwhh);
  offset(mx, Gdata.Wrf, lwhh, 1);                       // Set input w offset, lwhh
//                          Construct Plot and Draw

  row_vector data=ACQ.F(mx,N,Fstart,Fend,1.e-3);// Frequency acquisition
  GP_1D("Gauss.gnu", data, 0 , Fstart, Fend);           // Output in gnuplot
  GP_1Dplot("gnu.dat", "Gauss.gnu");                    // Interactive 1D gnuplot
  cout << "\n";
  }
```

GlutamicA.sys

```
SysName     (2) : Glutamic   - System Name (glutamic acid)
NSpins      (0) : 5          - Number of Spins in the System
Iso(0)      (2) : 1H         - Spin Isotope Type
Iso(1)      (2) : 1H         - Spin Isotope Type
Iso(2)      (2) : 1H         - Spin Isotope Type
Iso(3)      (2) : 1H         - Spin Isotope Type
Iso(4)      (2) : 1H         - Spin Isotope Type
PPM(0)      (1) : 4.295      - Chemical Shifts in PPM
PPM(1)      (1) : 2.092      - Chemical Shifts in PPM
PPM(2)      (1) : 1.969      - Chemical Shifts in PPM
PPM(3)      (1) : 2.314      - Chemical Shifts in PPM
PPM(4)      (1) : 2.283      - Chemical Shifts in PPM
J(0,1)      (1) : 4.6        - Coupling Constants in Hz
J(0,2)      (1) : 9.5        - Coupling Constants in Hz
J(0,3)      (1) : 0.0        - Coupling Constants in Hz
J(0,4)      (1) : 0.0        - Coupling Constants in Hz
J(1,2)      (1) : -14.7      - Coupling Constants in Hz
J(1,3)      (1) : 7.3        - Coupling Constants in Hz
J(1,4)      (1) : 7.3        - Coupling Constants in Hz
J(2,3)      (1) : 7.3        - Coupling Constants in Hz
J(2,4)      (1) : 7.3        - Coupling Constants in Hz
J(3,4)      (1) : -14.6      - Coupling Constants in Hz
Omega       (1) : 200        - Field Strength MHz (1H based)

            Gaussian Pulse Definitions

Note1: Two of {Gang, Gtime, GgamB1} used, {Gang, Gtime} preferentially
Note2: Either {Gspin} or {Gwrf, Giso} Sets Selectivity, {Gspin} preferentially
Note3: {Giso} Need not be set in a homonuclear system

Gstps       (0) : 41         - Steps over the Gaussian pulse
Gang        (1) : 90.0       - Pulse angle (degrees)
Glen        (1) : .1         - Pulse length (seconds)
Gcut        (1) : 0.025      - RF cutoff level (%GgamB1 -> 2.5%)
GW          (1) : 400.0      - Frequency at which to apply pulse
Giso        (2) : 1H         - Channel on which to apply pulse
Gphi        (1) : 0.0        - Phase on which to apply pulse
#GgamB1     (1) : 55.        - RF field strength (Hz)
```

GlutamicA2.sys

```
SysName     (2) : Glutamic   - System Name (glutamic acid)
NSpins      (0) : 5          - Number of Spins in the System
Iso(0)      (2) : 1H         - Spin Isotope Type
Iso(1)      (2) : 1H         - Spin Isotope Type
Iso(2)      (2) : 1H         - Spin Isotope Type
Iso(3)      (2) : 1H         - Spin Isotope Type
Iso(4)      (2) : 1H         - Spin Isotope Type
PPM(0)      (1) : 4.295      - Chemical Shifts in PPM
PPM(1)      (1) : 2.092      - Chemical Shifts in PPM
PPM(2)      (1) : 1.969      - Chemical Shifts in PPM
PPM(3)      (1) : 2.314      - Chemical Shifts in PPM
PPM(4)      (1) : 2.283      - Chemical Shifts in PPM
J(0,1)      (1) : 4.6        - Coupling Constants in Hz
J(0,2)      (1) : 9.5        - Coupling Constants in Hz
J(0,3)      (1) : 0.0        - Coupling Constants in Hz
J(0,4)      (1) : 0.0        - Coupling Constants in Hz
J(1,2)      (1) : -14.7      - Coupling Constants in Hz
J(1,3)      (1) : 7.3        - Coupling Constants in Hz
J(1,4)      (1) : 7.3        - Coupling Constants in Hz
J(2,3)      (1) : 7.3        - Coupling Constants in Hz
J(2,4)      (1) : 7.3        - Coupling Constants in Hz
J(3,4)      (1) : -14.6      - Coupling Constants in Hz
Omega       (1) : 200        - Field Strength MHz (1H based)

            Gaussian Pulse Definitions

Note1: Two of {Gang, Gtime, GgamB1} used, {Gang, Gtime} preferentially
Note2: Either {Gspin} or {Gwrf, Giso} Sets Selectivity, {Gspin} preferentially
Note3: {Giso} Need not be set in a homonuclear system

Gstps       (0) : 41         - Steps over the Gaussian pulse
Gang        (1) : 270.0      - Pulse angle (degrees)
Glen        (1) : .0001      - Pulse length (seconds)
Gcut        (1) : 0.025      - RF cutoff level (%GgamB1 -> 2.5%)
GW          (1) : 400.0      - Frequency at which to apply pulse
Giso        (2) : 1H         - Channel on which to apply pulse
Gphi        (1) : 270.0      - Phase on which to apply pulse
#GgamB1     (1) : 55.        - RF field strength (Hz)
```

Gpulse1.cc

```
/* Gpulse1.cc ****************************************************************-*-c++-*-
**                                                                                   **
**        NMR 1D Simulator Using A Gaussian Shaped Pulse                             **
**                                                                                   **
**  This program is an automated 1D NMR spectral simulator using                     **
**  shaped Gaussian pulses.  It runs inteactively, asking the user to                **
**  supply a parameter file filename as well as plot parameters.  The                **
**  system input is simply pulsed by the specified Gaussian.  It is a                **
**  modification from Gpulse0.cc in that it correctly uses the input                 **
**  pulse phase and uses the input pulse channel for the pulse/acquire               **
**  selectivity.                                                                     **
**                                                                                   **
**  This program does not include relaxation effects.  Also, the plot                **
**  is immediate in gnuplot, so this will die if gnuplot is that                     **
**  program is not accessible.                                                       **
**                                                                                   **
** Author:    S.A. Smith                                                             **
** Date:      7/8/96                                                                 **
** Update:    7/8/96                                                                 **
** Version:   3.5                                                                    **
**                                                                                   **
*************************************************************************************/

#include <gamma.h>                                  // Include GAMMA

main (int argc, char* argv[])

  {
  cout << "\n\n\t\t\tGAMMA 1D NMR Simulation Program";
  cout << "\n\t\t\tGaussian Pulses, No Relaxation\n\n";
//                          Read in System and Pulse Parameters

  int qn = 1;
  String filein;                                    // Input system file name
  query_parameter(argc, argv, qn++,                 // Get system file name
      "\n\tInput Parameter File? ", filein);
  sys_dynamic sys;                                  // A spin system
  sys.read(filein);                                 // Read in the system
  Gpuldat Gdata = read_Gpulse(filein, sys);         // Read Gaussian pulse params
  print_Gpulse(cout, Gdata);                        // Print Gaussian pusle params
  sys.offsetShifts(Gdata.Wrf);                      // Center system at pulse
//                    Determine Isotope Detection Type, Set Variables

 String IsoD = Gdata.Iso;
 gen_op sigma = sigma_eq(sys);                      // Set density matrix equilibrium
 gen_op H = Ho(sys);                                // Isotropic Hamiltonian
 gen_op detect = Fm(sys, IsoD);                     // Set detection operator to F-
 acquire1D ACQ(detect, H, 1.e-3);                   // No relaxation during acquisition (Ho)
 gen_op FXY = Fxy(sys, IsoD, Gdata.phi);            // For Gaussian RF Hamiltonian
 gen_op UGauss = Gpulse_U(H,FXY,Gdata);             // Gaussian pulse propagator
//                          Apply The Pulse Sequence

 sigma = evolve(sigma, UGauss);                     // Apply Gaussian pulse
 matrix mx = ACQ.table(sigma);                      // Set up 1D acquisition
//                          Set The Plotting Parameters

 double Fstart, Fend;                               // Number of points in FID
 query_parameter(argc, argv,                        // Get number of points
   qn++, "\n\tPlot Starting Frequency? ", Fstart);
 query_parameter(argc, argv,                        // Get number of points
   qn++, "\n\tPlot Final Frequency? ", Fend);
 int N;
 query_parameter(argc, argv,                        // Get number of points
   qn++, "\n\tPlot Points? ", N);
 double lwhh;                                       // Half-height linewidth
 query_parameter(argc, argv,                        // Get number of points
   qn++, "\n\tPlot Linewidths? ", lwhh);
 offset(mx, Gdata.Wrf, lwhh, 1);                    // Set input w offset, lwhh
//                          Construct Plot and Draw

 row_vector data=ACQ.F(mx,N,Fstart,Fend,1.e-3);// Frequency acquisition
 GP_1D("Gauss.gnu", data, 0 , Fstart, Fend);        // Output in gnuplot
 GP_1Dplot("gnu.dat", "Gauss.gnu");                 // Interactive 1D gnuplot
 cout << "\n";
 }
```

Gpulcorr2.cc

```
/* Gpulcorr3.cc *************************************************************************
**                                                                                   **
GAMMA Gaussian Pulse With Profiling Simulation                                       **
**                                                                                   **
** This program is an updated version of that published in the original             **
** GAMMA article, JMR 106A, 75-105 (1994).  Given an input spin system,             **
** a specified Gaussian pulse, and a few other parameters, this will                **
** calculate an NMR spectrum following the application of the pusle.                **
** At the same time, the Gaussian pulse exitation profile based on                  **
** response to a single spin 1/2 particle is computed.  This profile is             **
** used to phase correct the spectrum from the input spin system.  The              **
** phase corrected spectrum and the magnitude single spin profile are               **
** plotted to the screen using gnuplot.                                             **
**                                                                                   **
** This program is similar to Gpulcorr1.cc but it just outputs a single             **
** plot which contains the corrected spectrum and the single spin's                 **
** profile (magnitude mode).                                                        **
**                                                                                   **
** Author:    S.A. Smith                                                            **
** Date:      7/3/96                                                                **
** Last Date: 7/8/96                                                                **
** Copyright: S.A. Smith, July 1996                                                 **
** Limits:    1.) Needs >= GAMMA 3.5                                                **
**            2.) Uses Gnuplot Interactively                                        **
**            3.) No relaxation effects are included.                               **
**                                                                                   **
*************************************************************************************/

#include <gamma.h>                                  // Include GAMMA itself

main (int argc, char** argv)

  {
  cout << "\n\n\t\t\tGAMMA 1D NMR Simulation Program";
  cout << "\n\t\t     Gaussian Pulse Profile, No Relaxation\n\n";
//                       Read in System and Pulse Parameters
  int qn = 1;                                       // Query number
  String filein;                                    // Input system file name
  query_parameter(argc, argv, qn++,                 // Get system file name
        "\n\tInput Parameter File? ", filein);
  spin_system sys;                                  // A spin system
  sys.read(filein);                                 // Read in the system
  Gpuldat Gdata = read_Gpulse(filein, sys);         // Read Gaussian pulse params
  print_Gpulse(cout, Gdata);                        // Print Gaussian pusle params
  sys.offsetShifts(Gdata.Wrf);                      // Center system at pulse
//                 Determine Isotope Detection Type, Set Variables
  String IsoD = Gdata.Iso;                          // Set detection=pulse isotope
  gen_op sigma = sigma_eq(sys);                     // Set density op at equilibrium
  gen_op H = Ho(sys);                               // Isotropic Hamiltonian
  gen_op detect = Fm(sys, IsoD);                    // Set detection operator to F-
  acquire1D ACQ(detect, H, 1.e-3);                  // Set for acquisition, No rel.
  gen_op FXY = Fxy(sys, IsoD, Gdata.phi);           // For Gaussian RF Hamiltonian
  gen_op UGauss = Gpulse_U(H,FXY,Gdata);            // Gaussian pulse propagator
//                          Apply The Pulse Sequence
  sigma = evolve(sigma, UGauss);                    // Apply Gaussian pulse
  matrix mx = ACQ.table(sigma);                     // Perform 1D acquisition
//                          Set Plotting Parameters
  double Fst, Fend;                                 // Number of points in FID
  query_parameter(argc, argv, qn++,                 // Get number of points
     "\n\tPlot Starting Frequency? ", Fst);
  query_parameter(argc, argv,                       // Get number of points
   qn++, "\n\tPlot Final Frequency? ", Fend);
  int N;
  query_parameter(argc, argv,                       // Get number of points
           qn++, "\n\tPlot Points? ", N);
  double lwhh;                                      // Half-height linewidth
  query_parameter(argc, argv,                       // Get number of points
       qn++, "\n\tPlot Linewidths? ", lwhh);
  offset(mx, Gdata.Wrf, lwhh, 1);                   // Set input w offset, lwhh
//                      Calculate The Single Spin Profile
//                     (Generally Determine Phase Behavior)
  spin_system sys1(1);                              // Single spin system
  detect = Fm(sys1, IsoD);                          // Set detection operator to F-
  FXY = Fxy(sys1, IsoD, Gdata.phi);                 // For Gaussian RF Hamiltonian
  gen_op sigma0 = sigma_eq(sys1);                   // Initial density matrix
  row_vector profile(N);                            // Block for the profile
  double voff = Fst;                                // Begin @ 1st plotted frequency
  double delv = (Fend-Fst)/double(N-1);             // Offset increment
  complex z;
  for(int offs=0; offs<N; offs++)                   // Loop offsets
    {
    sys1.shift(0, voff);                            // Set spin chemical shift
    sys1.offsetShifts(Gdata.Wrf);                   // In pulse rotating frame
    H = Ho(sys1);                                   // Calculate the Hamiltonian
    UGauss = Gpulse_U(H, FXY, Gdata);               // Gaussian pulse propagator
    sigma = evolve(sigma0, UGauss);                 // Apply Gaussian pulse
    z = trace(detect, sigma);                       // Get magnetization
    profile.put(norm(z), offs);                     // Store magnitude
    voff += delv;                                   // Adjust the offset
    }
  GP_1D("profile.gnu",profile,0,Fst,Fend);          // Output in gnuplot
//        Perform the Phase Correction Using Phases for a 1-Spin System
//     Done For Each Transition in the Spin System and Corrected Discretely
```

```
 double w;
 complex zel;
 int ntr = mx.rows();                               // Number of transitions
 for(int tr=0; tr<ntr; tr++)                        // Loop the transitions
   {
   w = mx.getIm(tr,0)/(2.0*PI);                     // Transition frequency (Hz)
   sys1.shift(0, w);                                // Set spin chemical shift
   sys1.offsetShifts(Gdata.Wrf);                    // In pulse rotating frame
   H = Ho(sys1);                                    // Calculate the Hamiltonian
   UGauss = Gpulse_U(H, FXY, Gdata);                // Gaussian pulse propagator
   sigma = evolve(sigma0, UGauss);                  // Apply Gaussian pulse
   z = trace(detect, sigma);                        // Get transverse magnetization
   zel = mx.get(tr,1);                              // Large sytem transition intensity
   zel *= norm(z)/z;                                // Adjust transition phase
   mx.put(zel,tr,1);                                // Reset (adjusted) intensity
   }
 row_vector spec = ACQ.F(mx,N,Fst,Fend,1.e-3);      // Frequency acquisition
 GP_1D("spec.gnu",spec,0,Fst,Fend);                 // Output in gnuplot
```

//   Now Output the Corrected Spectrum & Profile Magnitude to Screen

```
 cout << "\n\n";                                    // Keep screen nice
 cout.flush();                                      // Also keeps screen nice
 ofstream gnuload("gnu.dat");                       // File of gnuplot commands
 gnuload << "set data style line\n";                // Set 1D plots to use lines
 gnuload << "set xlabel \"W(Hz)\"\n";               // Set X axis label
 gnuload << "set ylabel \"Intensity\"\n";           // Set Y axis label
 gnuload << "set title\"Spectrum\"\n";              // Set plot title
 gnuload << "plot \"spec.gnu\"";                    // Command to plot both
 gnuload << ", \"profile.gnu\"\n";                  // at the same time
 gnuload << "pause -1 \'<Return> To Exit \n";       // Pause before exit
 gnuload << "exit\n";                               // Now exit gnuplot
 gnuload.close();                                   // Close gnuplot command file
 system("gnuplot \"gnu.dat\"\n");                   // Invoke gnuplot now
 cout << "\n\n";                                    // Keep the screen nice
 }
```

Gpulpcorr2.cc

```
/* Gpulpcorr2.cc ***********************************************************************
**                                                                                   **
**           GAMMA Gaussian Pulse With 1st Order Phase Correction                     **
**                                                                                   **
** This program applies a Gaussian pulse to an arbitrary spin system.                 **
** The resulting spectrum is then adjusted by a common 1st order phase                **
** correction.  The idea is that we'd like to see how well the phase                  **
** corrections available in the spectrometer handle fixing the phase                  **
** problems resulting from selective Gaussian pulses.                                 **
**                                                                                   **
** Author:    S.A. Smith                                                              **
** Date:      7/9/96                                                                  **
** Last Date: 7/9/96                                                                  **
** Copyright: S.A. Smith, July 1996                                                   **
** Limits:    1.) Needs >= GAMMA 3.5                                                  **
**            2.) Uses Gnuplot Interactively                                          **
**            3.) No relaxation effects are included.                                 **
**                                                                                   **
**************************************************************************************/

#include <gamma.h>                                   // Include GAMMA itself

main (int argc, char** argv)

  {
  cout << "\n\n\t\t\tGAMMA 1D NMR Simulation Program";
  cout << "\n\t\t   Gaussian Pulse, 1st Order Phase Correction\n\n";
//                          Read in System and Pulse Parameters
  int qn = 1;                                        // Query number
  String filein;                                     // Input system file name
  query_parameter(argc, argv, qn++,                  // Get system file name
                   "\n\tInput Parameter File? ", filein);
  spin_system sys;                                   // A spin system
  sys.read(filein);                                  // Read in the system
  Gpuldat Gdata = read_Gpulse(filein, sys);          // Read Gaussian pulse params
  print_Gpulse(cout, Gdata);                         // Print Gaussian pusle params
  sys.offsetShifts(Gdata.Wrf);                       // Center system at pulse
//                     Determine Isotope Detection Type, Set Variables
  String IsoD = Gdata.Iso;                           // Set detection=pulse isotope
  gen_op sigma = sigma_eq(sys);                       // Set density op at equilibrium
  gen_op H = Ho(sys);                                // Isotropic Hamiltonian
  gen_op detect = Fm(sys, IsoD);                      // Set detection operator to F-
  acquire1D ACQ(detect, H, 1.e-3);                    // Set for acquisition, No rel.
  gen_op FXY = Fxy(sys, IsoD, Gdata.phi);            // For Gaussian RF Hamiltonian
  gen_op UGauss = Gpulse_U(H,FXY,Gdata);              // Gaussian pulse propagator
//                               Apply The Pulse Sequence
  sigma = evolve(sigma, UGauss);                     // Apply Gaussian pulse
  matrix mx = ACQ.table(sigma);                      // Perform 1D acquisition
//                               Set Plotting Parameters
  double Fst, Fend;                                  // Number of points in FID
  query_parameter(argc, argv, qn++,                  // Get number of points
    "\n\tPlot Starting Frequency? ", Fst);
  query_parameter(argc, argv,                         // Get number of points
   qn++, "\n\tPlot Final Frequency? ", Fend);
  int N;
  query_parameter(argc, argv,                        // Get number of points
             qn++, "\n\tPlot Points? ", N);
  double lwhh;                                       // Half-height linewidth
  query_parameter(argc, argv,                        // Get number of points
        qn++, "\n\tPlot Linewidths? ", lwhh);
  offset(mx, Gdata.Wrf, lwhh, 1);                    // Set input w offset, lwhh
  row_vector spec = ACQ.F(mx,N,Fst,Fend,1.e-3);      // Frequency acquisition
  GP_1D("spec.gnu",spec,0,Fst,Fend);                 // Output in gnuplot
//                   Calculate The 1st Order Phase Corrected Spectrum
  pcorrect(mx, Gdata.Wrf, Fend, 10);                 // Phase correct
  row_vector paspec=ACQ.F(mx,N,Fst,Fend,1.e-3);// Frequency acquisition
  GP_1D("paspec.gnu", paspec, 0, Fst, Fend);         // Output in gnuplot
//          Now Output the Spectrum & Phase Corrected Spectrum to Screen
  cout << "\n\n";                                    // Keep screen nice
  cout.flush();                                      // Also keeps screen nice
  ofstream gnuload("gnu.dat");                       // File of gnuplot commands
  gnuload << "set data style line\n";                // Set 1D plots to use lines
  gnuload << "set xlabel \"W(Hz)\"\n";               // Set X axis label
  gnuload << "set ylabel \"Intensity\"\n";           // Set Y axis label
  gnuload << "set title\"Spectrum\"\n";              // Set plot title
  gnuload << "plot \"spec.gnu\"";                    // Command to plot both
  gnuload << ", \"paspec.gnu\"\n";                   // at the same time
  gnuload << "pause -1 \'<Return> To Exit \n";       // Pause before exit
  gnuload << "exit\n";                               // Now exit gnuplot
  gnuload.close();                                   // Close gnuplot command file
  system("gnuplot \"gnu.dat\"\n");                   // Invoke gnuplot now
  cout << "\n\n";                                    // Keep the screen nice
  }
```

Gplot.cc

Generate Plots of Gaussian Pulse Waveforms

```
/* Gplot.cc *******************************************************************-*-c++-*-
**                                                                                    **
**  This program plots the rf-field amplitude versus time for                         **
**  given a Gaussian pulse as specified by four parameters:                           **
**                                                                                    **
**  1.) The field strength at maximum                                                 **
**  2.) The intensity cutoff (%) at the pulse endpoints                               **
**  3.) The number of steps to take for the pulse                                     **
**  4.) The time over which the pulse is active                                       **
**                                                                                    **
**  A gaussian function centered about time t  is given formally by                   **
**                                           o                                        **
**                                                                                    **
**                       [        2 /           2  ]                                  **
**             G(t) = exp | -(t-t ) / (2*sigma ) |                                    **
**                       [       0 /               ]                                  **
**                                                                                    **
**  The discrete function is similar except we would like to define                   **
**  sigma in terms of a cutoff.  That is to say, we should like to set                **
**  the Gaussian linewidth such that the first and last points are at                 **
**  a set percentage (of maximum == 1).                                               **
**                                                                                    **
**  For a cutoff of X%, we need to satisfy the following conditions                   **
**                                                                                    **
**                                  2          2                                      **
**                 0.0X = exp(-N / 8*sigma )                                          **
**                                                                                    **
**  or                                                                                **
**                 sigma = N / sqrt[-8*ln(0.0X)]                                      **
**                                                                                    **
**  where N is the number of Gaussian steps taken and N/2 is the peak                 **
**  maximum.  Setting the peak maximum to be related to an rf-field                   **
**  strength, this leaves us with the formula                                         **
**                                                                                    **
**                       [        2              2 ]                                  **
**             G(i) = exp | (2i-N) ln(0.0X)/ N  |                                     **
**                       [                         ]                                  **
**                                                                                    **
**  The output is sent directly to the screen using Gnuplot.  The user                **
**  may also have a plot output in Framemaker MIF format.                             **
**                                                                                    **
**                                                                                    **
**  Author:      S.A. Smith                                                           **
**  Date:        May 2 1995                                                           **
**  Last Update: May 8 1995                                                           **
**                                                                                    **
********************************************************************************** **/

#include <gamma.h>                                      // Include all of GAMMA

main(int argc, char* argv[])
  {
  int qn = 1;                                           // Query number
  int npts;                                             // Number of points
  double gamB1, time, fact;                             // Gaussian pulse parameters
  ask_Gpulse(argc, argv, qn,                            // Get pulse parameters
            npts, gamB1, time, fact, 1);
  row_vector G = Gvect(gamB1,npts,fact);                // Fill vector with waveform
  cout << “\n\n”;                                       // Keep screen nice
  GP_1D(“Gauss.gnu”, G);                                // Output rf lineshape gnuplot
  ofstream gnuload(“gnu.dat”);                          // File of gnuplot commands
  gnuload << “set data style line\n”;                   // Set 1D plots to use lines
  gnuload << “set xlabel \”time(sec)\”\n”;              // Set X axis label
  gnuload << “set ylabel \”gamB1(Hz)\”\n”;              // Set Y axis label
  gnuload << “set title\”Gaussian Pulse Shape\”\n”; // Set plot title
  gnuload << “plot \”Gauss.gnu\”\n”;                    // Command to plot
  gnuload << “pause -1 \’<Return> To Exit \n”;          // Command to pause
  gnuload << “exit\n”;                                  // Command to exit gnuplot
  gnuload.close();                                      // Close gnuplot command file
  system(“gnuplot \”gnu.dat\”\n”);                      // Now, actually run gnuplot
  String syn;                                           // When plot is complete, see
  cout << “\n\n\tFrameMaker Hardcopy[y/n]? “            ;// if output in FrameMaker is
  cin >> syn;                                           // desired.
  if(syn == “y”)
    FM_1D(“Gauss.mif”, G, 1);                           // Output to Framemaker
  cout << “\n\n”;                                       // Keep the screen nice
  }
```

Ghistplot.cc

Histogram Plots of Gaussian Pulse Waveforms

```
/* Ghistplot.cc **************************************************************-*-c++-*-
**                                                                          **
**  This program plots the rf-field amplitude versus time for a             **
**  given a Gaussian pulse as specified by four parameters:                 **
**                                                                          **
**  1.) The field strength at maximum                                       **
**  2.) The intensity cutoff (%) at the pulse endpoints                     **
**  3.) The number of steps to take for the pulse                           **
**  4.) The time over which the pulse is active                             **
**                                                                          **
**  A gaussian function centered about time t  is given formally by         **
**                                           o                              **
**                                                                          **
**                        [        2 /        2  ]                          **
**              G(t) = exp | -(t-t ) / (2*sigma ) |                         **
**                        [      0 /             ]                          **
**                                                                          **
**  The discrete function is similar except we would like to define         **
**  sigma in terms of a cutoff.  That is to say, we should like to set      **
**  the Gaussian linewidth such that the first and last points are at       **
**  a set percentage (of maximum == 1).                                     **
**                                                                          **
**  For a cutoff of X%, we need to satisfy the following conditions         **
**                                                                          **
**                              2
2                                                                           **
**                0.0X = exp(-N / 8*sigma )                                 **
**                                                                          **
**  or                                                                      **
**               sigma = N / sqrt[-8*ln(0.0X)]                              **
**                                                                          **
**  where N is the number of Gaussian steps taken and N/2 is the peak       **
**  maximum.  Setting the peak maximum to be related to an rf-field         **
**  strength, this leaves us with the formula                               **
**                                                                          **
**                        [       2          2 ]                            **
**              G(i) = exp | (2i-N) ln(0.0X)/ N  |                          **
**                        [                    ]                            **
**                                                                          **
**  Author:      S.A. Smith                                                 **
**  Date:        May 2 1995                                                 **
**  Last Update: May 2 1995                                                 **
**                                                                          **
*************************************************************************  */

#include <gamma.h>                                  // Include all of GAMMA

row_vector Gshape(double gamB1, double tau, int N, double fact=0.05)

   // Input      gamB  : The rf-field strength (Hz)
   //             tau  : Gaussian pulse length (sec)
   //              N   : Number of Gaussian steps
   //             fact : Cutoff factor
   // Output     angle: Gaussian pulse rotation angle
   //                         on resonance

 {
 if(fact>1.0 || fact<0.000001)                      // Make sure fact is between
   fact = 0.000001;                                 // [0,1]
 double tdiv = tau/double(N);                       // Incremental time
 double den = double((N-1)*(N-1));                  // Denominator
 double logf = log(fact);                           // Log of the cutoff factor
 double Z = logf/den;                               // Exponential factor
 double Gnorm;                                      // Normalized Gaussian intensity
 double num;
 int M = N+1;
 if(N > 100) M=0;
 row_vector Gshape(2*N+M);                          // Vector of Gaussian points
 double lastv, lastt;
 int I = 0;

 double Gs[N];
 for(int i=0; i<N; i++)                             // Loop over Gaussian steps
   {
   if(N-1-i < i)                                    //Use symmetry to avoid
     Gs[i] = Gs[N-1-i];                             //recalculating same pts
   else
     {
     num = double(2*i)-double(N-1);                 // Index so Gaussian mid-pulse
     Gnorm = exp(Z*num*num);                        // Normalize Gauss. amplitude
     Gs[i] = gamB1*Gnorm;                           // RF amplutude modulation
     }
   }

 double time = 0.0;                                 // Time in pulse
 for(i=0; i<N; i++)                                 // Loop over Gaussian steps
   {
   if(i)
     {
     Gshape.put(complex(time,Gs[i-1]),I++);         // For horizontals in hist.
     if(M) Gshape.put(complex(time,0),I++);         // For verticals in hist.
     }
   else if(M)
     Gshape.put(complex(time,0),I++);               // First vertical in hist.
   Gshape.put(complex(time,Gs[i]),I++);             // Gaussian intensity
   lastv = Gs[i];                                   // Store the previous intensity
   lastt = double(i);                               // Store the previous point
```

```
    if(i==N-1)                                      // For the last point
      {
      time += tdiv;
      Gshape.put(complex(time,Gs[i]),I++);          // For last horizontal in hist.
      if(M) Gshape.put(complex(time,0),I++);        // For last vertical in hist.
      }
    time += tdiv;
    }                                               // (evolve/acq step goes here)
  return Gshape;
  }


main(int argc, char* argv[])
  {
  int qn = 1;                                       // Query number
  int npts;                                         // Number of points
  query_parameter(argc, argv, qn++,                 // Get number of steps(pts)
   "\n\tNumber of Points in Gaussian? ", npts);
  if(npts < 2) npts = 2048;
  double gamB1;
  query_parameter(argc, argv, qn++,                 // Get rf-field strength
   "\n\tRF-Field Stength (Hz)? ", gamB1);
  double time;
  query_parameter(argc, argv, qn++,                 // Get pulse length
   "\n\tGaussian Pulse Length (sec)? ", time);
  double fact;
  query_parameter(argc, argv, qn++,                 // Get system file name
   "\n\tPercent Intensity at Ends [0, 1]? ", fact);
  cout << "\n\n";                                   // Keep the screen nice
  row_vector G = Gshape(gamB1,time,npts,fact);
  GP_xy("Gauss.gnu", G);                            // Output rf lineshape gnuplot
  ofstream gnuload("gnu.dat");                      // File of gnuplot commands
  gnuload << "set data style line\n";               // Set 1D plots to use lines
  gnuload << "plot \"Gauss.gnu\"\n";                // Plot IxA in gnuplot
  gnuload << "pause -1 \'<Return> To Exit \n";      // Plot IxA in gnuplot
  gnuload << "exit\n";                              // Plot IxA in gnuplot
  gnuload.close();                                  // Close gnuplot command file
  system("gnuplot \"gnu.dat\"\n");                  // Plot to screen
  cout << "\n\n";                                   // Keep the screen nice
  }
```

5 WALTZ

5.1 Overview

The module ***PulWALTZ***, which contains the class ***WALTZ***, facilitates the use of WALTZ pulse cycles in GAMMA nmr simulations. Class WALTZ contains parameters which define how WALTZ cycles is to be implemented and provides functions for building WALTZ based waveforms, composite pulses, and pulse trains.

5.2 Chapter Contents

5.2.1 WALTZ Section Listing

5.2.2 WALTZ Function Listing

5.2.3 WALTZ Figures & Tables Listing

5.2.4 WALTZ Examples

5.3 Constructors and Assignment

5.3.1 WALTZ

Usage:

```
#include <PulWALTZ.h>
WALTZ()
WALTZ(double gB1, const String& ch, double ph=0, double off=0);
WALTZ(const WALTZ& WP)
```

Description:

The function *WALTZ* is used to create a WALTZ parameter container.

1. PulWALTZ() - Creates an “empty” NULL WALTZ parameter. Can be later filled by an assignment.
2. PulWALTZ(double gB1, const& ch, double ph, double off) - Sets up WALTZ for having an rf-field strength of ***gB1*** Hz on the channel specified by ***ch***. WALTZ will be applied with an overall phase of ***ph*** degrees and an offset of ***off*** Hz.
3. PulWALTZ(const PulWALTZ &PWF1) - Constructs an identical PulWALTZ to the inputPWF1.

Return Value:

WALTZ returns no parameters. It is used strictly to create a WALTZ parameter container.

Examples:

```
PulWALTZ PW;                              // Empty WALTZ parameters
PulWALTZ PW1(538.9, “13C”);               // WALTZ @ γB1=538.9 on 13C channel
PulWALTZ PW3(PW1);                        // Another WALTZ identical to PW1
```

See Also: =

5.3.2 =

Usage:

```
#include <PulWALTZ.h>
void WALTZ operator = (PulWALTZ &PWF1)
```

Description:

The unary operator = (the assignment operator) allows for the setting of one WALTZ to another WALTZ. If the WALTZ being assigned to exists it will be overwritten by the assigned WALTZ.

Return Value:

None, the function is void

Example:

```
PulWALTZ PW1(538.9, “13C”);               // WALTZ @ γB1=538.9 on 13C channel
PulWALTZ PW2 = PW1;                       // Another WALTZ identical to PW1
```

See Also: PulWALTZ

5.4 Pulse Waveform Functions

5.4.1 WF

5.4.2 WF_WALTZR

5.4.3 WF_WALTZK

5.4.4 WF_WALTZQ

Usage:

```
#include <PulWALTZ.h>
PulWaveform WALTZ::WF()
PulWaveform WALTZ::WF_WaltzR()
PulWaveform WALTZ::WF_WaltzK()
PulWaveform WALTZ::WF_WaltzQ()
```

Description:

The functions ***WF_WALTZR***, ***WF_WALTZK***, and ***WF_WALTZQ***, will return a composite pulses for WALTZ-R, WALTZ-K, and WALTZ-Q respectively. The function ***WF*** is identical to ***WF_WALTZR.***

Return Value:

The function returns an composite pulse.

Example:

```
WALTZ WP;                               // Declare WALTZ Parameters
WP.read("filein.pset")                  // Read in WALTZ Parameters
PulWaveform WR = WP.WFp(sys);           // WALTZ-R waveform
WR = WP.PCmpWALTZR(sys);                // Also WALTZ-R
PulWaveform WK = WP.WF_WALTZK(sys);     // WALTZ-K waveform
PulWaveform WQ= WP.WF_WALTZQ(sys);      // WALTZ-Q waveform
```

See Also: WALTZ-4, WALTZ-8, WALTZ-16

5.5 Composite Pulse Functions

5.5.1 PCmp

5.5.2 PCmpWALTZR

5.5.3 PCmpWALTZK

5.5.4 PCmpWALTZQ

Usage:

```
#include <PulWALTZ.h>
PulComposite WALTZ::PCmp(const spin_system&)
PulComposite WALTZ::PCmpWaltzR(const spin_system&)
PulComposite WALTZ::PCmpWaltzK(const spin_system&)
PulComposite WALTZ::PCmpWaltzQ(const spin_system&)
```

Description:

The functions ***PCmpWALTZR***, ***PCmpWALTZK***, and ***PCmpWALTZQ***, will return a composite pulses for WALTZ-R, WALTZ-K, and WALTZ-Q respectively. The function ***PCmp*** is identical to ***PCmpWALTZR.***

Return Value:

The function returns an composite pulse.

Example:

```
spin_system sys;                                // Declare a spin system
sys.read(“filein.sys”);                         // Read in the spin system
WALTZ WP;                                       // Declare WALTZ Parameters
WP.read(“filein.pset”)                          // Read in WALTZ Parameters
PulComposite WR = WP.PCmp(sys);                 // WALTZ-R composite pulse
WR = WP.PCmpWALTZR(sys);                        // Also WALTZ-R
PulComposite WK = WP.PCmpWALTZK(sys);           // WALTZ-K composite pulse
PulComposite WQ= WP.PCmpWALTZQ(sys);            // WALTZ-Q composite pulse
```

See Also: CycWALTZ-4, CycWALTZ-8, CycWALTZ-16

5.6 Pulse Cycle Functions

5.6.1 CycWALTZ4

5.6.2 CycWALTZ8

5.6.3 CycWALTZ16

Usage:

```
#include <PulWALTZ.h>
PulCycle WALTZ::CycWaltz4(const spin_system&)
PulCycle WALTZ::CycWaltz8(const spin_system&)
PulCycle WALTZ::CycWaltz16(const spin_system&)
```

Description:

The functions ***CycWALTZ4***, ***CycWALTZ8***, and ***CycWALTZ16***, will return a pulse cycles for WALTZ-4, WALTZ-8, and WALTZ-16 respectively.

Return Value:

The function returns an pulse cycle.

Example:

```
spin_system sys;                        // Declare a spin system
sys.read(“filein.sys”);                 // Read in the spin system
WALTZ WP;                               // Declare WALTZ Parameters
WP.read(“filein.pset”)                  // Read in WALTZ Parameters
PulCycle W4 = WP.CycWALTZ4(sys);        // WALTZ-4 pulse cycle
PulCycle W8 = WP.CycWALTZ8(sys);        // WALTZ-8 pulse cycle
PulCycle W16 = WP.CycWALTZ16(sys);      // WALTZ-16 pulse cycle
```

See Also: PCmpWALTZ-K, PCmpWALTZ-R, PCmpWALTZ-Q

5.7 Input Functions

5.7.1 read

Usage:

```
#include <PulWALTZ.h>
void WALTZ::read(const String& filename, int idx = -1)
void WALTZ::read(ParameterAVLSet& pset, int idx = -1)
```

Description:

The function read will fill a WALTZ parameter with values in either a specified external ASCII file ***filename***, or from a specfied parameter set ***pset***. If a non-negative index is also included as an argument then the parameters that define the returned WALTZ will be assumed indicated with a “(#)” on each where # is the values of ***idx***. See the section on this chapter discussing how to specify WALTZ related parameters in an external file.

Return Value:

void.

Example:

```
WALTZ WP;                         // Declare WALTZ Parameters
WP.read(“filein.pset”)            // Read in WALTZ Parameters
ParameterAVLSet pset;             // A GAMMA parameter set
pset.read(“another.pset”);        // Read in a parameter set
WP.read(pset, 3);                 // Set WALTZ from 3rd in pset
```

See Also: ask_read

5.7.2 ask_read

Usage:

```
#include <PulWALTZ.h>
void WALTZ::ask_read(int argc, char* argv[], int argn)
```

Description:

The function read will interactively set thefill a WALTZ parameter with values in either a specified external ASCII file ***filename***, or from a specfied parameter set ***pset***. If a non-negative index is also included as an argument then the parameters that define the returned WALTZ will be assumed indicated with a “(#)” on each where # is the values of ***idx***. See the section on this chapter discussing how to specify WALTZ related parameters in an external file.

Return Value:

void.

Example:

```
WALTZ WP;                                // Declare WALTZ Parameters
WP.read(“filein.pset”)                   // Read in WALTZ Parameters
ParameterAVLSet pset;                    // A GAMMA parameter set
pset.read(“another.pset”);               // Read in a parameter set
WP.read(pset, 3);                        // Set WALTZ from 3rd in pset
```

See Also: ask_read

5.8 Output Functions

5.8.1 print

Usage:

```
#include <PulWALTZ.h>
ostream& WALTZ::print(ostream& ostr)
```

Description:

The function ***print*** will output WALTZ parameters into a specified output stream ***ostr***.

Return Value:

void.

Example:

```
WALTZ WP;                    // Declare WALTZ Parameters
WP.read("filein.pset")       // Read in WALTZ Parameters
WP.print(cout);              // Send WALTZ values to screen
```

See Also: <<

5.8.2 <<

Usage:

```
#include <PulWALTZ.h>
ostream& WALTZ::print(ostream& ostr)
```

Description:

The operator **<<** will output WALTZ parameters to standard output.

Return Value:

void.

Example:

```
WALTZ WP;                    // Declare WALTZ Parameters
WP.read("filein.pset")       // Read in WALTZ Parameters
cout << WP;                  // Send WALTZ values to screen
```

See Also: print

5.9 Description

5.9.1 Introduction

The functions in module ***PulWALTZ*** and Class ***WALTZ*** (contained in module ***PulWALTZ***), is designed to facilitate the use of WALTZ[1] pulse trains in GAMMA NMR simulation programs. In GAMMA, as in an NMR experiment, we should like to use WALTZ pulse trains as individual steps in a general pulse sequence, including use in variable delays as part of multi-dimensional experiments and/or use in pulse trains during acquisition steps.

5.9.2 WALTZ Parameters

A variable of type WALTZ contains only primitive parameters. In particular, it contains the four values which define a pulse waveform: 1.) A ***# steps***, 2.) An ***rf-field strength***, 3.) An ***rf-phase*** 4.) An ***rf-offset***. These parameters can be used to completely determine how to set up composite pulses such as that used in a WALTZ pulse train.

5.9.3 WALTZ Waveforms & Composite Pulses

The simplest WALTZ sequence is based on a 3 step composite 180 pulse. The pulses are applied with the same rf-strength but vary in their applied length and phase. The details are shown in the following figure, the composite pulse called WALTZ-R.

WALTZ-R 3 Step Sequence: Composite 180 Pulse

Step	Angle	Phase
1	90.0	0.0
2	$\overline{180.0}$	$\overline{180.0}$
3	270.0	0.0

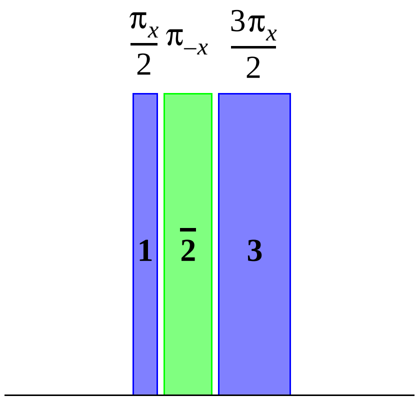


Figure 7-1 The basic WALTZ 3-step waveform. The blue steps indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table). Shorthand notation is used for this sequence, 1$\bar{2}$3, where each integer reflects multiples of a 90 pulse and bar indicates the phase shift.

The WALTZ-R composite pulse is used to build WALTZ-4 pulse cycles. Additional composite pulses are used in other WALTZ sequences. The next is WALTZ-K, a five step composite pulse as shown in the next figure.

1.

WALTZ-K 5 Step Sequence

Step	Angle	Phase
1	180.0	180.0
2	360.0	0.0
3	180.0	180.0
4	270.0	0.0
5	90.0	180.0

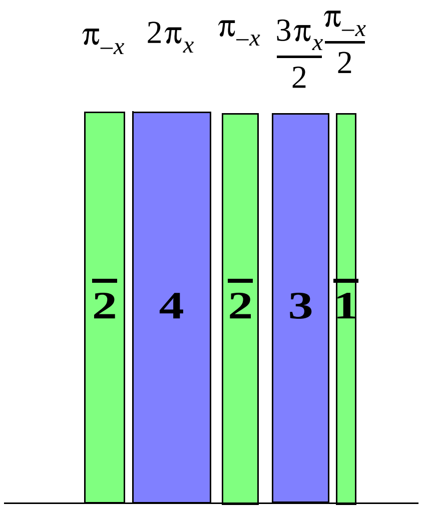


Figure 7-1 The basic WALTZ-K 5-step waveform. The blue steps indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table). Shorthand notation is used for this sequence, $\bar{2}4\bar{2}3\bar{1}$, where each integer reflects multiples of a 90 pulse and bar indicates the phase shift.

WALTZ-Q composites are used in making WALTZ-16 pulse cycles.

WALTZ-Q 9 Step Sequence

Step	Angle	Phase
1	270.0	180.0
2	360.0	0.0
3	180.0	180.0
4	270.0	0.0
5	90.0	180.0
6	180.0	0.0
7	360.0	180.0
8	180.0	0.0
9	270.0	180.0

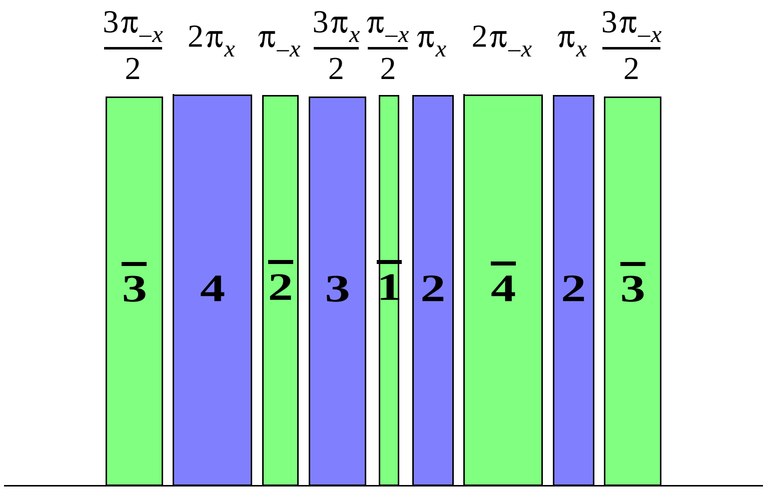


Figure 7-1 The basic WALTZ-Q 9-step waveform. The blue steps indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table). Shorthand notation is used for this sequence, $\bar{3}4\bar{2}3\bar{1}2\bar{4}2\bar{3}$, where each integer reflects multiples of a 90 pulse and bar indicates the phase shift.

5.9.4 WALTZ Pulse Cycles

The simplest WALTZ based pulse cycle is WALTZ-4. This sequence cycles the basic composite 180 sequence, WALTZ-R, through a 4-step phase change. Using "R" to represent the WALTZ-R base composite pulse, the base WALTZ-4 cycle sequence is given by RR$\overline{RR}$ where $\overline{R}$ implies a 180 phase shifted R. This relationship is shown in the figure below.

WALTZ-4 Pulse Cycle

Step	Angle	Phase	Type	Cycle
1	90.0	0.0	1	R
2	180.0	180.0	$\bar{2}$	
3	270.0	0.0	3	
4	90.0	0.0	1	R
5	180.0	180.0	$\bar{2}$	
6	270.0	0.0	3	
7	90.0	180.0	$\bar{1}$	$\overline{R}$
8	180.0	0.0	2	
9	270.0	180.0	$\bar{3}$	
10	90.0	180.0	$\bar{1}$	$\overline{R}$
11	180.0	0.0	2	
12	270.0	180.0	$\bar{3}$	

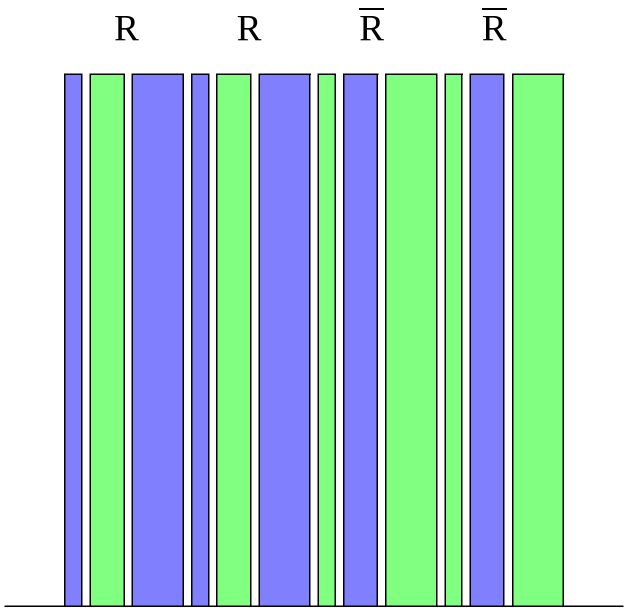


Figure 7-2 The WALTZ-4 pulse cycle. Four WALTZ-R composite pulses are linked together with the last two having their phase angle shifted by 180 degrees. Using R to designate the WALTZ-R and a bar to indicate a 180 phase shift, WALTZ-4 can be described as repeating RR$\overline{RR}$. The blue steps above indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table).

There are two important properties of periodic decoupling sequences in an ideal situation. The first is that their performance is unaffected by phase inversion. The second is that they are unaffected by a cyclic permutation of some part of the sequence. However, in actuality there are small residual effects from choosing one possibility over the other. To compensate, one simply couples sequences together with varying phase and permutation.

The WALTZ-8 sequence cycles the 5 step composite WALTZ-K through a 4-step phase change. Using "K" to represent the WALTZ-K composite pulse, the WALTZ-8 cycle sequence is given by K$\overline{KK}$K where $\overline{K}$ implies a 180 phase shifted K. This relationship is shown in the figure below.

WALTZ-8 Pulse Cycle

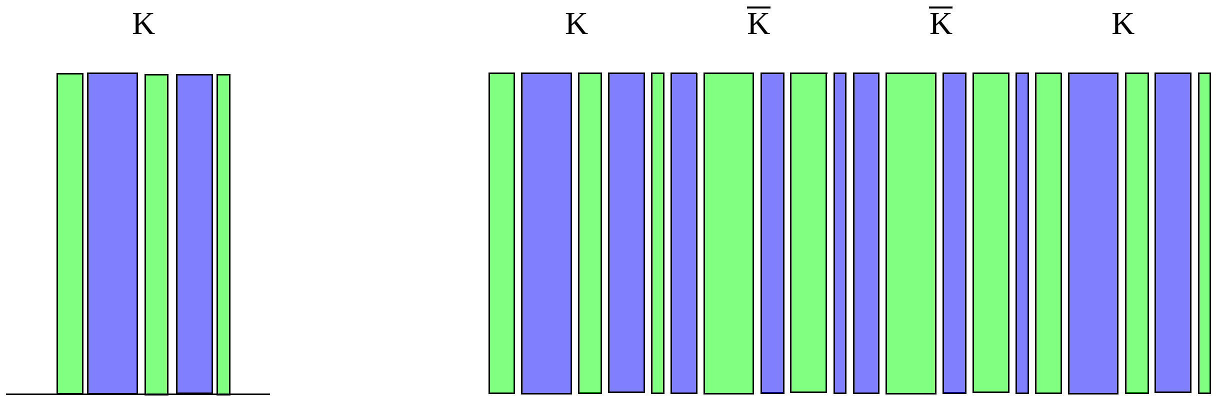


Figure 7-1 The basic 5-step WALTZ-K waveform (on the left) is repeated 4 times to produces the WALTZ-8 cycle. The blue steps indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table). Shorthand notation is used for this sequence, $K\overline{K}\overline{K}K$. These four steps are repeated as long as WALTZ-8 is to be applied.

The WALTZ-16 sequence cycles the 9 step composite WALTZ-Q through a 4-step phase change. Using “Q” to represent the WALTZ-Q composite pulse, the WALTZ-16 cycle sequence is given by $Q\overline{Q}\overline{Q}Q$ where $\overline{Q}$ implies a 180 phase shifted Q. This relationship is shown in the figure below.

WALTZ-16 Pulse Cycle

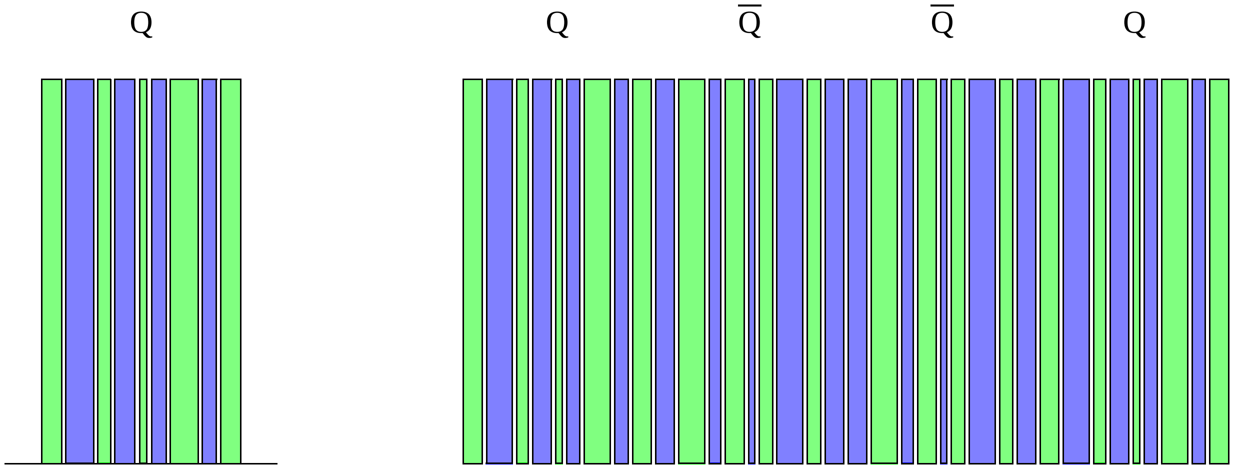


Figure 7-1 The basic 9-step WALTZ-Q waveform (on the left) is repeated 4 times to produces the WALTZ-16 cycle. The blue steps indicate pulses that are applied without any phase shift whereas the green step indicates a pulse with a 180 phase shift (as reflected in the table). Shorthand notation is used for this sequence, $Q\overline{Q}\overline{Q}Q$. These four steps are repeated as long as WALTZ-16 is to be applied.

5.9.5 WALTZ Parameters

This section describes how an ASCII file may be constructed that is self readable by a WALTZ variable. The file can be created with an editor of the users choosing and is read with the WALTZ member function “read”. This provides for an extremely flexible and program independent means of implementing WALTZ in NMR simulations.

The WALTZ (ASCII) input file is scanned for the specific parameters which specify the pulse-delay parameters[1]: rf-strength, rf-phase, rf-offset, and the rf-channel. These parameters are recognized by certain keywords, as shown in the following table.

Table 1: Spin System Parameters

Parameter Keyword	Assumed Units	Examples Parameter (Type) : Value - Statement
WALTZgamB1	Hz	WALTZgamB1 (1) : 600.0 - Field Strength (Hz)
WALTZiso	none	WALTZiso (2) : 19F - WALTZ rf pulse channel
WALTZphi	degrees	WALTZphi (1) : 2.0 - WALTZ rf phase (deg)

The order in which these parameters reside in the ASCII file is of no consequence.

The format of each parameter is quite simple and general for all GAMMA parameters. At the beginning of a line the parameter keyword is written followed by an optional index number in parenthesis. This is then followed by one or more blanks and then an integer in parentheses. The integer corresponds to the type of parameter value: 0 = integer, 1 = floating point, or 2 = string. Following the parenthesis should be at least one blank then a colon to indicate the parameter value follows. The parameter value is then written followed by some blanks then a hyphen followed by an optional comment.

There is one major restriction; keywords and string parameters cannot contain blanks. For example, v (0) is unknown, v(0) is. The string value 19 F is unknown, 19F is fine. If multiple WALTZ pulse-delay steps need to be defined in the same file then simply put an index on all parameters associated with a desired WALTZ and read the parameters using that index.

To read the file, see the documentation for function read (or ask-read) and look at the example programs in this chapter along with their input files.

Channel: WALTZiso

This parameter is optional. It will define which isotope channel the WALTZ rf-pulse will be applied on. If no channel is specified GAMMA will assume that all spins in the system being treated are affected by the rf. Thus if no channel is specified and WALTZ is utilized in an NMR simulation the system should be homonuclear or the same WALTZ should be desired on all channels (same offset, phase, etc.)

1. Note that the ASCII file must contain viable parameters in GAMMA format. Indeed, the file is a GAMMA parameter set and, as such, may contain any amount of additional information along with the valid WALTZ parameters.

Channel: WALTZphi

This parameter is optional. It will define the rf-phase of the WALTZ pulse. If no phase is specified it will be taken to be zero.

Pulse Strength: WALTZgamB1

The parameters { WALTZang, WALTZgamB1, WALTZtp } work together. WALTZgamB1 se the pulse strength if either WALTZang or WALTZgamB1 have also been specified. If only WALTZgamB1 is specified amongst the three an error will result when reading these parameters to define WALTZ. If all three parameters have been specified then WALTZgamB1 will be ignored, the strength set by { WALTZang, WALTZtp }

5.10 WALTZ Examples

5.10.1 Reading WALTZ Parameters

To keep GAMMA programs using WALTZ sequences versatile, users will want to keep all WALTZ specifications undetermined in the code. As the program runs, WALTZ settings are either specified interactively and/or read in from an external ASCII (parameter) file. This section gives examples of the latter case. The figure below shows an ASCII parameter file on the left and some GAMMA program code on the right.

Reading WALTZ Parameters

```
WALTZphi        (1) : 90.0    - WALTZ overall rf-phase (deg)
WALTZiso        (2) : 1H      - WALTZ pulse channel
WALTZgamB1      (1) : 983.0   - WALTZ pulse strength (Hz)
WALTZiso(3)     (2) : 19F     - WALTZ pulse channel
WALTZgamB1(3)(2) : 2045.9 - WALTZ pulse strength (Hz)
```

```
WALTZ WP;                         // Declare WALTZ parameters
WP.read(“WALTZ.pset”);            // Read WALTZ from file
WP.read(“WALTZ.pset”, 3);         // Read WALTZ from file
WP.ask_read(argc, argv, 1);       // Read WALTZ from file
WP.ask_read(argc, argv, 2, 3);    // Read WALTZ from file
```

Figure 7-2 Typical WALTZ parameters (left) and the GAMMA code which reads them. The parameters are contained in an ASCII file which the code reads to set up use of WALTZ.

The ASCII parameter file (on the left) is taken to be called “WALTZ.pset” and is read in by the program code. Thus one can change the WALTZ parameters independent of the GAMMA code. The ASCII file format is typical of GAMMA parameter sets: The line ordering is of no consequence, the column spacing is not important, the end “- comments” can be left off, and additional lines of text or parameters may be included.

The GAMMA code is color coded with the parameters they read in the previous figure. Thus the second line (blue) will read the blue parameters and set up WALTZ with a strength of 983 Hz on the proton channel with an overall phase of 90 degrees. Similarly, the next line (green) will read the parameters colored green from the same ASCII file but sets up WALTZ with a strength of 2.0459 kHz on the 19F channel (no phase, no offset).

The next line will interactively ask the user to supply a filename where the program can get to WALTZ parameters. This filename (in this case “WALTZ.pset”) will be prompted for unless the user specifies the file on the command line when the program is executed. The following line does the same but reads the WALTZ parameters indexed with a “3” from the file.

Using a combination of these commands, the user has complete flexibility in defining one or more WALTZ sequences in the same GAMMA program. The WALTZ parameters can be easily changed by either changing their values in the ASCII file and/or changing the filename given to the program. See the section of WALTZ parameters to see which parameters can be used in setting up WALTZ sequences. See the other programs in this chapter for full examples GAMMA programs using them.

5.10.2 WALTZ Decoupling

In this section we shall produce a simple 1D NMR spectrum under WALTZ-16 decoupling. A hard 90 pulse will be applied to a chosen spin system on the acquisition channel. Then WALTZ-16 will be applied on the decoupler channel during acquisition. The resulting FID will be apodized and Fourier transformed, the NMR spectrum put on screen using Gnuplot. Note that relaxation and exchange effects will be ignored in this simulation. The code for simple WALTZ-16 decoupling is shown below:

```
/* WALTZdec0.cc *********************************************************************
**
**                                                                                  **
**                     GAMMA Decoupling Example Program                             **
**                                                                                  **
**  This program uses the class WALTZ to perform a simple decoupling                **
**  simulation.  A hard ideal pulse will be applied to an input spin                **
**  system.  Subsequently, an acquisition will be performed with WALTZ-16           **
**  decoupling applied on a specified channel.                                      **
**                                                                                  **
** Author:      S.A. Smith                                                          **
** Date:        3/9/98                                                              **
** Update:      3/9/98                                                              **
** Version:     3.5.4                                                               **
** Copyright:   S. Smith.  Modify this program as you see fit for personal          **
**              use, but you must leave the program intact if redistrubuted         **
**                                                                                  **
*************************************************************************************
**/

#include <gamma.h>
#
main(int argc, char* argv[])
 {
 cout << "\n\t\t\t\tWALTZ Decoupling\n\n";
 int qn = 1;                                       // Query index
 spin_system sys;                                  // Declare a spin system
 WALTZ WP;                                         // WALTZ parameters
 String filename = sys.ask_read(argc,argv,qn++);   // Ask for/read in the system
 cout << sys;                                      // Have a look (for setting SW)
 WP.read(filename);                                // Read in WALTZ parameters
 PulCycle PCyc = WP.CycWALTZ16(sys);               // Construct WALTZ-16 cycle
 String IsoD = sys.symbol(0);                      // Detection/pulse channel
 if(sys.heteronuclear())                           // If heteronuclear system
   query_parameter(argc, argv, qn++,               // ask for detection channel
         "\n\tDetection Isotope? ", IsoD);
 double SW;                                        // Spectral width
 query_parameter(argc, argv, qn++,                 // Get desired spectral width
         "\n\tSpectral Width (Hz)? ", SW);
 int npts = 1024;                                  // Block size (must be base 2)
 query_parameter(argc, argv, qn++,                 // Get block size
            "\n\tBlock Size? ", npts);
 double lwhh = 1.0;                                // Half-height linewidth
 query_parameter(argc, argv, qn++,                 // Ask for apodization strength
          "\n\tApodization (Hz)? ", lwhh);
 double td = 1/SW;                                 // Set dwell time
 double R = (lwhh/2)*HZ2RAD;                       // Set apodization rate
 double tt = (npts-1)*td;                          // Total FID length
 gen_op H = Ho(sys);                               // Set isotropic Hamiltonian
 gen_op Det = Fm(sys, IsoD);                       // Set detection operator to F-
 gen_op sigma0 = sigma_eq(sys);                    // Set density mx equilibrium
 gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);     // This is 1st 90 pulse
 row_vector data = PCyc.FID(npts,td,Det,sigmap);   // Perform acquisition under WALTZ-16
 row_vector exp = Exponential(npts,tt,0.0,R,0);    // Here is an exponential
 row_vector fidap = product(data,exp);             // Apodized FID
 data = FFT(fidap);                                // Transformed FID -> spectrum
 GP_1D("spec.asc", data, 0, -SW/2, SW/2);          // Output ASCII file
 GP_1Dplot("spec.gnu", "spec.asc");                // Plot to screen using Gnuplot
 }
```

The first half of this program simply sets the parameters up interactively. Note that both the spin system and the WALTZ parameters are contained in the same file who's name the user must specify.

The second half of the program does the simulation. The FID is acquired using the pulse cycle function "FID" and the pulse cycle has been set to WALTZ-16. The last few lines apodize and transform the FID then spit the plot out on the screen.

Addition of relaxation & exchange effects and/or changing the 1st pulse to non-ideal will require only minor modifications of a few lines.

The input parameter (ASCII) file is listed on the following page. It contains parameters for both the spin system and the WALTZ settings.

Example of a WALTZ decoupling input file (WALTZdec.sys)

```
SysName      (2) : WALTZ    - Name of the Spin System
NSpins       (0) : 4        - Number of Spins in the System
Iso(0)       (2) : 1H       - Spin Isotope Type
Iso(1)       (2) : 1H       - Spin Isotope Type
Iso(2)       (2) : 1H       - Spin Isotope Type
Iso(3)       (2) : 13C      - Spin Isotope Type
v(0)         (1) : 105.0    - Chemical Shifts in Hz
v(1)         (1) : -174.32  - Chemical Shifts in Hz
v(2)         (1) : 15.0     - Chemical Shifts in Hz
v(3)         (1) : 0.0      - Chemical Shifts in Hz
J(0,1)       (1) : 10.0     - Coupling Constants in Hz
J(0,2)       (1) : 7.9      - Coupling Constants in Hz
J(0,3)       (1) : 22.0     - Coupling Constants in Hz
J(1,2)       (1) : 2.8      - Coupling Constants in Hz
J(1,3)       (1) : 32.0      - Coupling Constants in Hz
J(2,3)       (1) : 18.3     - Coupling Constants in Hz
Omega        (1) : 400      - Spect. Freq. in MHz (1H based)

WALTZphi     (1) : 0        - WALTZ pulse phase (deg)
WALTZiso     (2) : 13C      - WALTZ pulse channel
WALTZgamB1   (1) : 1500.0   - WALTZ pulse strength (Hz)
```

When the program (WALTZdec0.cc) is compiled its execution will produce a plot on screen if the Gnuplot program is available. Assuming the executable is called a.out, the following command will produces a spectrum:

a.out WALTZdec.sys 1H 500 1024 .5

The command “a.out” alone will prompt you for input values. Had you input the above command (or parameters) the spectrum should appear as shown in the following figure.

^{13}C Decoupled Spectrum Using WALTZ-16

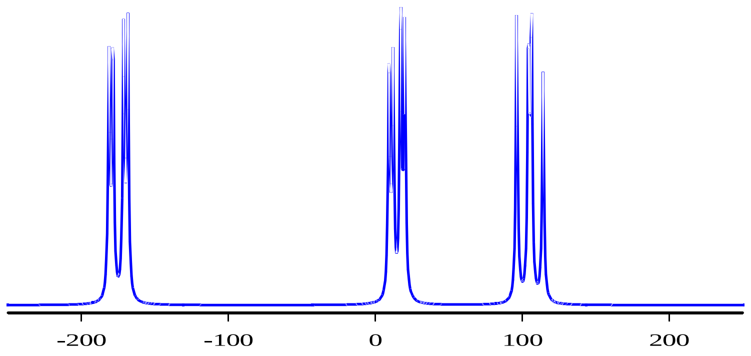


Figure 7-3 The spectrum produced using the program WALTZdec0.cc with input parameter file WALTZdec.sys. The decoupler was applied to the 13C channel with a 1.5 kHz field strength. Detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed with a 0.5 Hz line-broadening.

By either editing the input file or specifying a different input file, the spectrum can be radically altered. For example, by setting the WALTZ pulse strength to zero one obtains the following spectrum.

^{13}C Coupled Spectrum, Zero Strength WALTZ-16

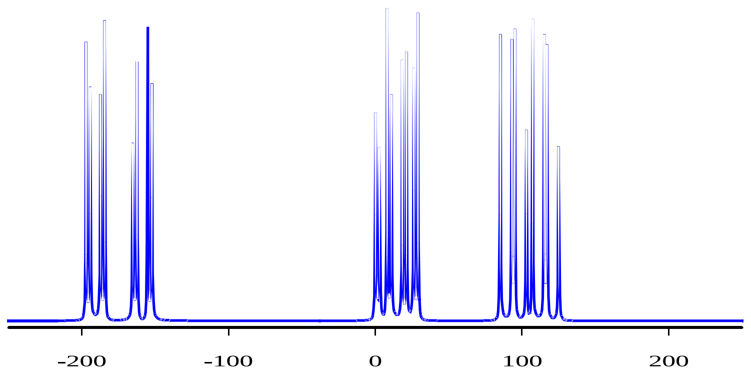


Figure 7-4 Same as previous figure but with zero decoupler field.

5.10.3 WALTZ-16 Decoupling vs. Field

We can readily modify the previous program to loop over differing rf-field strengths and determine how well WALTZ-16 does at decoupling. In this case we will just read in a series of gB1 values from an external ASCII file and loop over them producing a 1D spectrum at each value. We'll spit out all the spectra in a single stack plot.

```
/* WALTZdecstk1.cc *****************************************************************
**                                                                               **
**                  GAMMA Decoupling Test Program                                **
**                                                                               **
**  This program uses the class WALTZ to perform a simple decoupling             **
**  simulation.  A hard ideal pulse will be applied to an input spin             **
**  system.  Subsequently, an acquisition is taken with WALTZ-16                 **
**  decoupling applied on a specified channel.  This pulse-acquisiton            **
**  process will be repeated over a series of specified decoupler                **
**  rf-field strengths.  The decoupled spectra will be output on screen          **
**  if Gnuplot is available.  The stack plot is also output in                   **
**  FrameMaker MIF format.                                                       **
**                                                                               **
** Author:      S.A. Smith                                                       **
** Date:        3/11/98                                                          **
** Update:      3/11/98                                                          **
** Version:     3.5.4                                                            **
** Copyright:   S. Smith.  You can modify this program as you see fit            **
**              for personal use, but you must leave the program intact          **
**              if you re-distribute it.                                         **
**                                                                               **
***********************************************************************************/

#include <gamma.h>

main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tWALTZ Decoupling Vs. Decoupler Strength\n\n";
  int qn = 1;                                       // Query index
  spin_system sys;                                  // Declare a spin system
  WALTZ WP;                                         // WALTZ parameters
  String filename;                                  // Input filename
  filename = sys.ask_read(argc,argv,qn++);          // Ask for/read in the system
  cout << sys;                                      // Have a look (for setting SW)
  WP.read(filename);                                // Read in WALTZ parameters
  cout << WP;
  PulCycle PCyc;
  query_parameter(argc, argv, qn++,                 // Ask for field strength file
    "\n\tFile of Field Strengths? ", filename);
  int N;                                            // Number of field strengths
  double* gB1s = GetDoubles(filename, N);           // Get array of field strengths

  String IsoD = sys.symbol(0);                      // Detection/pulse channel
  if(sys.heteronuclear())                           // If heteronuclear system
    query_parameter(argc, argv, qn++,               // ask for detection channel
          "\n\tDetection Isotope? ", IsoD);
  double SW;                                        // Spectral width
  query_parameter(argc, argv, qn++,                 // Get desired spectral width
          "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                  // Block size (must be base 2)
  query_parameter(argc, argv, qn++,                 // Get block size
              "\n\tBlock Size? ", npts);
  double lwhh = 3.0;                                // Half-height linewidth
  query_parameter(argc, argv, qn++,                 // Ask for apodization strength
           "\n\tApodization (Hz)? ", lwhh);
  double td = 1/SW;                                 // Set dwell time
  double R = (lwhh/2)*HZ2RAD;                       // Set apodization rate
  double tt = (npts-1)*td;                          // Total FID length
  gen_op H = Ho(sys);                               // Set isotropic Hamiltonian
  gen_op Det = Fm(sys, IsoD);                       // Set detection operator to F-
  gen_op sigma0 = sigma_eq(sys);                    // Set density mx equilibrium
  gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);     // This is 1st 90 pulse
  row_vector data, exp, fidap;
  matrix datamx(N,npts);
  for(int i=0; i<N; i++)
    {
    WP.strength(gB1s[i]);                           // Set WALTZ field strength
    PCyc = WP.CycWALTZ16(sys);                      // Set WALTZ-16 pulse cycle
    data = PCyc.FID(npts,td,Det,sigmap);            // Perform acquisition
    exp = Exponential(npts,tt,0.0,R,0);             // Here is an exponential
    fidap = product(data,exp);                      // Apodized FID
    data = FFT(fidap);                              // Transformed FID -> spectrum
    datamx.put_block(i,0,data);
    }
  double Nm1 = double(N-1);
  String AF("stk.asc");
  String GF("stk.gnu");
  GP_stack(AF, datamx, 0,1,N,0.0,Nm1);
  GP_stackplot(GF, AF);
  FM_stack("stk.mif", datamx, 1.5, 1.5, 1);
  }
```

The modifications from the previous program are obvious. An external ASCII file is used to specify a list of decoupler field strengths and these are read into the program. These fields are looped over, a new spectrum computed at each decoupler strength.

The spectra are put into a matrix which is given to the Gnuplot routines for display as a stack plot on screen. In addition, the stack plot is output in FrameMaker MIF format for incorporation into documents in an editable form. The latter is shown in the following figure.

^{13}C WALTZ-16 Decoupling Versus RF-Field Strength

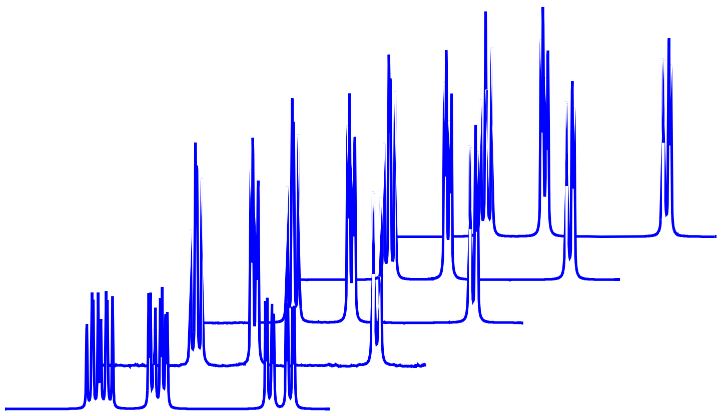


Figure 7-5 Proton spectra produced using the program WALTZdec1.cc with input parameter file WALTZdec.sys and decoupler strength file WALTZdecBs. The decoupler was applied to the 13C channel with field strengths shown. Detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed with a 1.5 Hz line-broadening. Note: the baseline “noise” exists because of my use of an asynchronous acquisition. By setting a spectral width that is commensurate with the WALTZ-16 cycle length that will disappear (see function FIDsync, its used in the next example).

When the program (WALTZdecstk1.cc) is compiled its execution will produce a stack plot on screen if the Gnuplot program is available. Assuming the executable is called a.out, the following command will produce the plot shown in the previous figure:

a.out WALTZdec.sys WALTZdecBs 1H 500 1024 1.5

The ASCII file WALTZdecBs contains a list of rf-field strengths (in Hz) that the program used. The file has a single field strength per line and is shown next.

WALTZ decoupling rf-field input file (WALTZdecBs)

```
0
200
400
600
800
```

Unlike GAMMA parameter set files (such as WALTZdec.sys) this file is simple ASCII and cannot have anything other than a single floating point or integer value per line. No additional comments may be included.

5.10.4 WALTZ Types vs. Decoupling

For something different, lets compare how well the different WALTZ sequences perform. The base WALTZ composite 180 pulse, WALTZ-R, is cycled to produce WALTZ-4. The base composite pulse is altered to produce WALTZ-8 and WALTZ-16, which are based on WALTZ-K and WALTZ-Q composite pulses respectively. The variations are improvements which are meant to suppress artifacts that can result from pulse mis-calibration.

In this example we will generate WALTZ decoupled spectra using all six of these sequences. Since we are using “perfect” rectangular pulses and neglecting relaxation, the resulting spectra should be (nearly) identical. Note the difference in the use of Composite Pulses (WALTZ-R, WALTZ-K, WALTZ-Q) relative to Pulse Cycles (WALTZ-4, WALTZ-8, and WALTZ-16). Their associated functionality in GAMMA is very similar although the cycles are automatically accounting for the phase changes in the composite pulses. Also, note that more obvious differences between the sequences should arise when relaxation effects are include and/or we intentionally mis-set the pulse lengths.

^{13}C WALTZ Decoupling Versus WALTZ Type

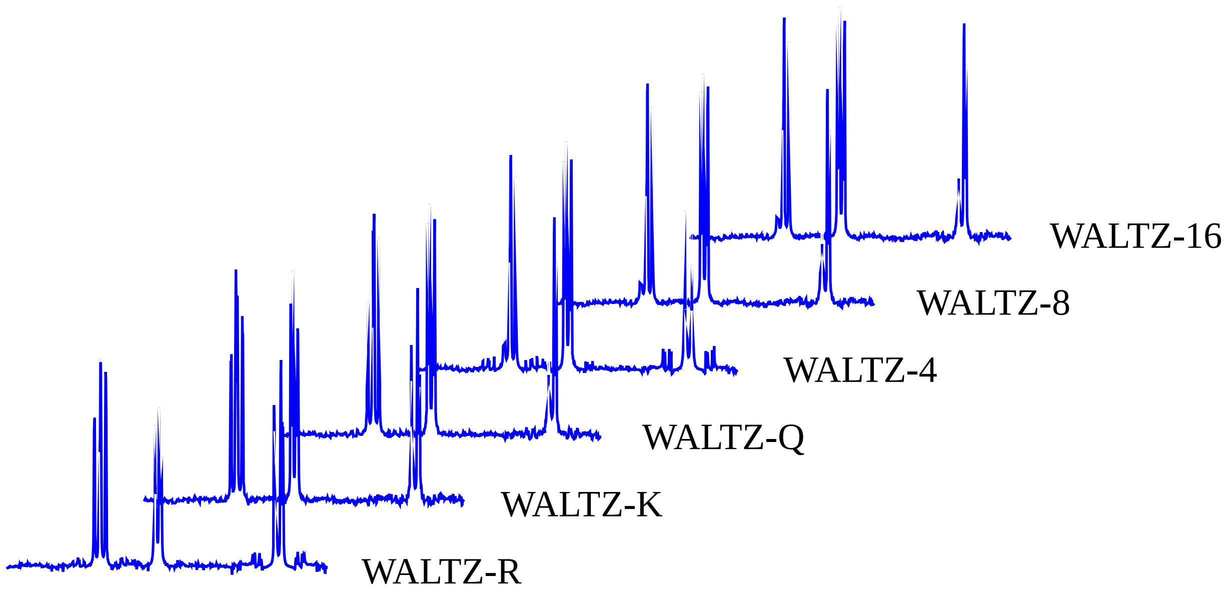


Figure 7-6 Proton spectra produced using the program WALTZtypes1.cc with input parameter file WALTZdec.sys. The decoupler was applied to the 13C channel and detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed with a 1.0 Hz line-broadening. Note: the baseline “noise” exists because of my use of an asynchronous acquisition. By setting a spectral width that is commensurate with the various WALTZ sequence lengths that will disappear (see function FIDsync).

The program below produced the previous stack plot.

```
/* WALTZtypes1.cc ********************************************************************
**                                                                                  **
**                        GAMMA Decoupling Test Program                             **
**                                                                                  **
**  This program uses the class WALTZ to perform a simple decoupling                **
**  simulation.  A hard ideal pulse will be applied to an input spin                **
**  system.  Subsequently, an acquisition is taken with one of the                  **
**  WALTZ decoupling sequences applied on a specified channel.  This                **
**  pulse-acquisition process will be repeated over a series of WALTZ               **
**  sequences (WALTZ-{R,K,Q,4,8,16}).  The decoupled spectra will be                **
**  output on screen if Gnuplot is available.  The stack plot is also               **
**  output in FrameMaker MIF format.                                                **
**                                                                                  **
** Author:      S.A. Smith                                                          **
** Date:        3/11/98                                                             **
** Version:     3.5.4                                                               **
**                                                                                  **
*************************************************************************************/

#include <gamma.h>

main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tWALTZ Decoupling Vs. WALTZ Type\n\n";
  int qn = 1;                                       // Query index
  spin_system sys;                                  // Declare a spin system
  WALTZ WP;                                         // WALTZ parameters
  String filename;                                  // Input filename
  filename = sys.ask_read(argc,argv,qn++);          // Ask for/read in the system
  cout << sys;                                      // Have a look (for setting SW)
  WP.read(filename);                                // Read in WALTZ parameters
  cout << WP;

  String IsoD = sys.symbol(0);                      // Detection/pulse channel
  if(sys.heteronuclear())                           // If heteronuclear system
    query_parameter(argc, argv, qn++,               // ask for detection channel
            "\n\tDetection Isotope? ", IsoD);
  double SW;                                        // Spectral width
  query_parameter(argc, argv, qn++,                 // Get desired spectral width
            "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                  // Block size (must be base 2)
  query_parameter(argc, argv, qn++,                 // Get block size
                "\n\tBlock Size? ", npts);
  double lwhh = 3.0;                                // Half-height linewidth
  query_parameter(argc, argv, qn++,                 // Ask for apodization strength
             "\n\tApodization (Hz)? ", lwhh);
  double td = 1/SW;                                 // Set dwell time
  double R = (lwhh/2)*HZ2RAD;                       // Set apodization rate
double tt = (npts-1)*td;                              // Total FID length
gen_op H = Ho(sys);                                   // Set isotropic Hamiltonian
gen_op Det = Fm(sys, IsoD);                           // Set detection operator to F-
gen_op sigma0 = sigma_eq(sys);                        // Set density mx equilibrium
gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);         // This is 1st 90 pulse
row_vector data, exp, fidap;
int N = 3;                                            // Number of WALTZ cycle types
int M = 3;                                            // Number of WALTZ composite types
matrix datamx(N+M,npts);
PulComposite Comps[M];
Comps[0] = WP.PCmpWALTZR(sys);                        // Set WALTZ-R composite pulse
Comps[1] = WP.PCmpWALTZK(sys);                        // Set WALTZ-K composite pulse
Comps[2] = WP.PCmpWALTZQ(sys);                        // Set WALTZ-Q composite pulse
PulCycle Cycles[N];
Cycles[0] = WP.CycWALTZ4(sys);                        // Set WALTZ-4 pulse cycle
Cycles[1] = WP.CycWALTZ8(sys);                        // Set WALTZ-8 pulse cycle
Cycles[2] = WP.CycWALTZ16(sys);                       // Set WALTZ-16 pulse cycle
PulComposite PComp;
PulCycle PCyc;
int i;
for(i=0; i<M; i++)
  {
  PComp = Comps[i];
  data = PComp.FID(npts,td,Det,sigmap);               // Perform acquisition
  exp = Exponential(npts,tt,0.0,R,0);                 // Here is an exponential
  fidap = product(data,exp);                          // Apodized FID
  data = FFT(fidap);                                  // Transformed FID -> spectrum
  datamx.put_block(i,0,data);
  }
for(int j=0; i<N+M; i++, j++)
  {
  PCyc = Cycles[j];
  data = PCyc.FID(npts,td,Det,sigmap);                // Perform acquisition
  exp = Exponential(npts,tt,0.0,R,0);                 // Here is an exponential
  fidap = product(data,exp);                          // Apodized FID
  data = FFT(fidap);                                  // Transformed FID -> spectrum
  datamx.put_block(i,0,data);
  }
double Nm1 = double(N+M-1);
String AF("stk.asc");
String GF("stk.gnu");
GP_stack(AF, datamx, 0,1,N+M,0.0,Nm1);
GP_stackplot(GF, AF);
FM_stack("stk.mif", datamx, 1.5, 1.5, 1);
}
```

5.10.5 WALTZ Decoupling with Relaxation

How can we include the effects of relaxation (and/or exchange) when we decouple? Since we already have a couple of programs that simulate decoupled spectra under WALTZ without relaxation, we need only make the proper modifications to them and their input files in order to obtain the simulation(s) we want.

Lets review a few of basic changes we'll need. First, rather than working with an isotropic spin system (spin_system) in our program, we need to work with a oriented spin system that is moving isotropically. That is, a spin system that keeps track of dipolar, CSA, and quadrupolar tensors for all spins or spin-pairs. Thus we need to replace ***spin_system*** with ***sys_dynamic***. Second, when the system is read in from an external ASCII file it will look for tensor quantities as well as dynamical values (correlation times). Next we will have to create a relaxation matrix and Liouvillian that defines how the system evolves. And lastly, we'll have to use an FID function that includes that evolves under the defined Liouvillian so that relaxation (and exchange) are accounted for.

This seems like it is complicated, but in fact amounts only to about few lines of code changes.In this example, I'll modify the previous program to include relaxation.... but I'll remove the loop over the different WALTZ types and just allow the user to choose one. A couple of results are shown in the next figure and we'll see the code after that.

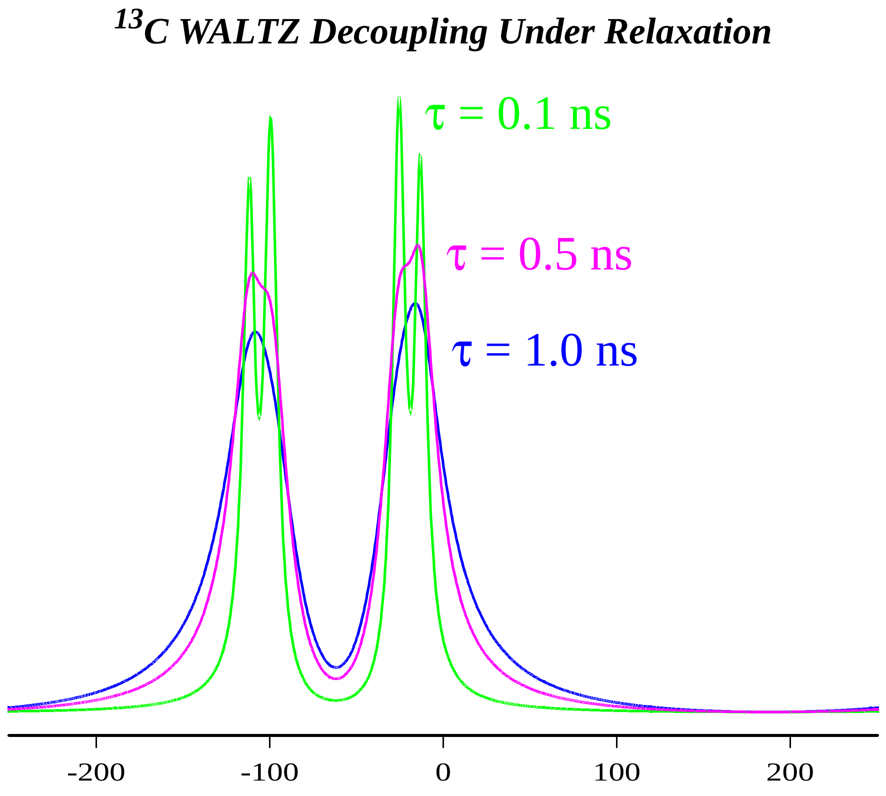


Figure 7-7 Proton spectra produced using the program WALTZdec2.cc with input parameter file WALTZdec3.dsys. The decoupler was applied to the 13C channel and detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed without line-broadening. The input file is given on a subsequent page, WALTZdec3.dsys. Successive plots were made with the same input file except for the correlation times (kept spherical) altered as reflected on the plot.

```
/* WALTZdec2.cc ***********************************************************************
**                                                                                   **
**                   GAMMA Decoupling Test Program                                   **
**                                                                                   **
**  This program uses the class WALTZ to perform a simple decoupling                 **
**  simulation.  A hard ideal pulse will be applied to an input spin                 **
**  system.  Subsequently, an acquisition is performed with WALTZ                    **
**  decoupling applied on a specified channel.  The decoupled spectrum               **
**  will be output on screen if Gnuplot is available.  The spectrum                  **
**  is also output in FrameMaker.MIF format.                                         **
**                                                                                   **
**                                                                                   **
** Note: This program is identical to WALTZdec1.cc except that it                    **
** includes the effects of relaxation.                                               **
**                                                                                   **
** Author:      S.A. Smith                                                           **
** Date:        3/30/98                                                              **
** Update:      3/30/98                                                              **
** Version:     3.5.4                                                                **
** Copyright:   S. Smith.  You can modify this program as you see fit                **
**              for personal use, but you must leave the program intact              **
**              if you re-distribute it.                                             **
**                                                                                   **
**************************************************************************************/

#include <gamma.h>
main(int argc, char* argv[])
 {
 cout << "\n\t\t\t\tWALTZ Decoupling With Relaxation\n\n";
 int qn = 1;                                   // Query index
 sys_dynamic sys;                              // Declare a spin system
 WALTZ WP;                                     // WALTZ parameters
 String filename;                              // Input filename
 filename = sys.ask_read(argc,argv,qn++);      // Ask for/read in the system
 cout << sys;                                  // Have a look (for setting SW)
 WBRExch WBR;                                  // Relaxation parameters
 WBR.read(filename, sys);                      // Read in relaxation parameters
 cout << WBR;                                  // Have a look at relaxation settings
 WP.read(filename);                            // Read in WALTZ parameters

 cout << "\n\tWALTZ Decoupling Schemes:\n";    // Ask fo WALTZ type
 cout << "\n\t\tRepeated WALTZ-R (1)";
 cout << "\n\t\tRepeated WALTZ-K (2)";
 cout << "\n\t\tRepeated WALTZ-Q (3)";
 cout << "\n\t\tWALTZ-4 Cycle (4)";
 cout << "\n\t\tWALTZ-8 Cycle (5)";
 cout << "\n\t\tWALTZ-16 Cycle (6)";
 int wt;
 query_parameter(argc, argv, qn++,             // Get number of steps
            "\n\n\tWALTZ Type? ", wt);
 if(wt<1 || wt>6) wt=1;                        // Insure [1,6]
```

```
PulComposite PCmp;
PulCycle PCyc;
switch(wt)
 {
 case 1: PCmp = WP.PCmpWALTZR(sys); break;
 case 2: PCmp = WP.PCmpWALTZK(sys); break;
 case 3: PCmp = WP.PCmpWALTZQ(sys); break;
 case 4: PCyc = WP.CycWALTZ4(sys); break;
 case 5: PCyc = WP.CycWALTZ8(sys); break;
 case 6: PCyc = WP.CycWALTZ16(sys); break;
 }
String IsoD = sys.symbol(0);                          // Detection/pulse channel
if(sys.heteronuclear())                               // If heteronuclear system
 query_parameter(argc, argv, qn++,                    // ask for detection channel
      "\n\tDetection Isotope? ", IsoD);
double SW;                                            // Spectral width
query_parameter(argc, argv, qn++,                     // Get desired spectral width
      "\n\tSpectral Width (Hz)? ", SW);
int npts = 1024;                                      // Block size (must be base 2)
query_parameter(argc, argv, qn++,                     // Get block size
         "\n\tBlock Size? ", npts);
double lwhh = 3.0;                                    // Half-height linewidth
query_parameter(argc, argv, qn++,                     // Ask for apodization strength
      "\n\tApodization (Hz)? ", lwhh);
double td = 1/SW;                                     // Set dwell time
double R = (lwhh/2)*HZ2RAD;                           // Set apodization rate
double tt = (npts-1)*td;                              // Total FID length
gen_op H = Ho(sys);                                   // Set isotropic Hamiltonian
gen_op Det = Fm(sys, IsoD);                           // Set detection operator to F-
gen_op sigma0 = sigma_eq(sys);                        // Set density mx equilibrium
gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);         // This is 1st 90 pulse
super_op L = WBR.REX(sys,H);                          // Get relaxation superoperator
row_vector data;
if(wt<=3)
 {
 PCmp.setRelax(sys,L);                                // Put relaxation into pulse
 data=PCmp.FIDR(npts,td,Det,sigmap,1);                // Perform acquisition
 }
else
 {
 PCyc.setRelax(sys,L);                                // Put relaxation into pulse
 data=PCyc.FIDR(npts,td,Det,sigmap,1);                // Perform acquisition
 }
row_vector exp=Exponential(npts,tt,0.0,R,0);          // Here is an exponential
row_vector fidap = product(data,exp);                 // Apodized FID
data = FFT(fidap);                                    // Transformed FID -> spectrum
GP_1D("spec.asc", data, 0, -SW/2, SW/2);              // Output ASCII file
GP_1Dplot("spec.gnu", "spec.asc");                    // Plot to screen using Gnuplot
FM_1D("spec.mif", data,14,14,-SW/2, SW/2);            // Plot in FrameMaker MIF
}
```

The previous simulation was given the input file "WALTZdec3.dsys" which is shown below. This file contains a spin system, WALTZ parameters, and relaxation parameters.

```
SysName      (2) : WALTZ   - Name of the Spin System
NSpins       (0) : 3       - Number of Spins in the System
Iso(0)       (2) : 1H      - Spin Isotope Type
Iso(1)       (2) : 1H      - Spin Isotope Type
Iso(2)       (2) : 13C     - Spin Isotope Type
v(0)         (1) : 105.0   - Chemical Shifts in Hz
v(1)         (1) : 20.0    - Chemical Shifts in Hz
v(2)         (1) : 0.0     - Chemical Shifts in Hz
J(0,1)       (1) : 12.0    - Coupling Constants in Hz
J(0,2)       (1) : 22.0    - Coupling Constants in Hz
J(1,2)       1) : 28.0     - Coupling Constants in Hz
Coord(0)     (3) : ( 0.0,  0.0,  0.0)     - Coordinate Point (A)
Coord(1)     (3) : ( 0.0,  0.0,  1.1)     - Coordinate Point (A)
Coord(2)     (3) : ( 0.0,  0.0,  -1.1)    - Coordinate Point (A)
Taus         (3) : ( 0.5,  0.5,  0.5)   - Correlation Times (ns)
Omega        (1) : 400     - Spec. Freq. in MHz (1H based)

WALTZphi     (1) : 0       - WALTZ pulse phase (deg)
WALTZiso     (2) : 13C     - WALTZ pulse channel
WALTZgamB1   (1) : 3000.0  - WALTZ pulse strength (Hz)

Rlevel       (0) : 4        - Relaxation Computation Level
Rtype        (0) : 0        - Relaxation Computation Type
RDD          (0) : 1        - Dipolar Relaxation Flag
RDDdfs       (0) : 0        - Dipolar DFS Flag
RCC          (0) : 0        - CSA Relaxation Flag
RCCdfs       (0) : 0        - CSA DFS Flag
RQQ          (0) : 0        - Quad Relaxation Flag
RQQdfs       (0) : 0        - Quad DFS Flag
RDQ          (0) : 0        - Dip-Quad Relaxation Flag
RDQdfs       (0) : 0        - Quad DFS Flag
RDC          (0) : 0        - Dip-CSA Relaxation Flag
RDCdfs       (0) : 0        - Dip-CSA DFS Flag
RQC          (0) : 0        - Quad-CSA Relaxation Flag
RQCdfs       (0) : 0        - Quad-CSA DFS Flag
```

For successive simulations (as shown in the previous figure) the input correlation times were altered. The three lines used were

```
Taus         (3) : ( 0.1,  0.1,  0.1)   - Correlation Times (ns)
Taus         (3) : ( 0.5,  0.5,  0.5)   - Correlation Times (ns)
Taus         (3) : ( 1.0,  1.0,  1.0)   - Correlation Times (ns)
```

In all three cases, WALTZ-16 was used. Each spectrum size was set to 1024 points and no line-broadening was used in the processing. The spectral width was set to 500 Hz and detection was on the proton channel.

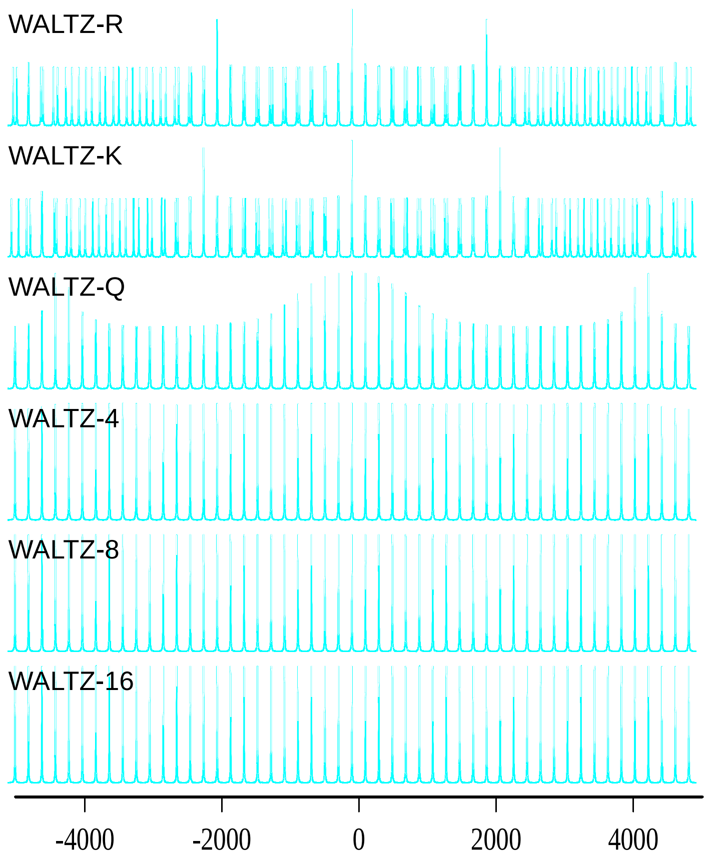
5.10.6 WALTZ Decoupling Profile

In this section we shall attempt to produce a WALTZ decoupling profile. A hard 90 pulse will be applied to a simple heteronuclear spin system on the acquisition channel. Then WALTZ decoupling will be applied on the decoupler channel during acquisition. The user will specify which WALTZ sequence to use. The resulting FID will be apodized and Fourier transformed. This pulse-delay process will be repeated for differing offsets on the decoupler channel. Each spectrum will be plotted with its center at the offset frequency to produce the profile.

The really no significant differences between this and our previous calculations. To determine a profile one typically uses the simplest spin system (here a two spin heteronuclear system). The 1D spectrum is recalculated after either moving the decoupler rf offset or, equivalently, moving all decoupler isotope channel chemical shifts. The spectra are all just put into a single vector, offset so their respective centers are set to be referenced to the decoupler offset value.

Here are some of the results from the GAMMA program given on the following page.

WALTZ Decoupling Profiles



```
SysName      (2) : WALTZp - Name of the Spin System
NSpins       (0) : 2      - Number of Spins
Iso(0)       (2) : 1H     - Spin Isotope Type
Iso(1)       (2) : 13C    - Spin Isotope Type
v(0)         (1) : 0.0    - Chemical Shifts in Hz
v(1)         (1) : 0.0    - Chemical Shifts in Hz
J(0,1)       (1) : 221.0  - Coupling Constants in Hz
Omega        (1) : 720    - Spec. Freq. in MHz

WALTZphi     (1) : 0      - WALTZ phase (deg)
WALTZiso     (2) : 13C    - WALTZ channel
WALTZgamB1   (1) : 5000.0 - WALTZ strength (Hz)
```

The above lines constitute the input file used, WALTZprof.sys The program was run using the command

```
a.out WALTZprof.sys 6 50 1024 1.5 25 200
```

where a.out is the program executable and “6” was varied from 1-6 on successive runs. The output was displayed on the screen and placed into FrameMaker MIF format, the latter of which has been placed into this document as seen on the left.

Figure 7-8 WALTZ decoupling profiles produced from the program WALTZprof2.cc. Decoupling was performed on the carbon channel in a ^{13}C-^{1}H two spin system. The decoupling rf-field strength was set to 5 kHz and the scalar coupling to 221 Hz. A linebroadening of 1.5 Hz was used in processing the spectra. The block size was 1K and the offset increment set to 200 Hz.

```
/* WALTZprof2.cc ********************************************************************
**                                                                                 **
**                        GAMMA Decoupling Test Program                            **
**                                                                                 **
**  This program uses the class WALTZ to perform a simple decoupling               **
**  simulation.  A hard ideal pulse will be applied to a simple two                **
**  spin  heteronuclear system.  Subsequently, an acquisition will be              **
**  performed with WALTZ decoupling applied on one the channel which              **
**  is not being detected.  This process will be repeated over a range             **
**  of decoupler offsets.  The result is a WALTZ decoupler profile.                **
**  The profile will be plotted on screen if Gnuplot is available on               **
**  the system.  The profile will also be output in FrameMaker MIF.                **
**                                                                                 **
** Author:      S.A. Smith                                                         **
** Date:        4/7/98                                                             **
** Update:      4/7/98                                                             **
** Version:     3.5.4                                                              **
**                                                                                 **
************************************************************************************/

#include <gamma.h>

main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tWALTZ Decoupling Profile\n\n";
//                       Read In Spin System & WALTZ Parameters
  int qn = 1;                                         // Query index
  spin_system sys;                                    // Declare a spin system
  WALTZ WP;                                           // WALTZ parameters
  String filename;                                    // Input filename
  filename = sys.ask_read(argc,argv,qn++);            // Ask for/read in the system
  cout << sys;                                        // Have a look (for setting SW)
  if(sys.spins()!=2 || sys.homonuclear())
    cout << "\n\tWarning! This program has been"
         << " set up for a two spin heteronuclear"
         << " system.  Results on other systems"
         << " can be unpredictable........";
  WP.read(filename);                                  // Read in WALTZ parameters
//                           Set WALTZ Decoupling Scheme
cout << "\n\tWALTZ Decoupling Schemes:\n";            // Ask fo WALTZ type
  cout << "\n\t\tRepeated WALTZ-R (1)";
  cout << "\n\t\tRepeated WALTZ-K (2)";
  cout << "\n\t\tRepeated WALTZ-Q (3)";
  cout << "\n\t\tWALTZ-4 Cycle (4)";
  cout << "\n\t\tWALTZ-8 Cycle (5)";
  cout << "\n\t\tWALTZ-16 Cycle (6)";
  int wt;
  query_parameter(argc, argv, qn++,                   // Get number of steps
                "\n\n\tWALTZ Type? ", wt);
  if(wt<1 || wt>6) wt=1;                              // Insure [1,6]

  PulComposite PCmp;
  PulCycle PCyc;
  switch(wt)
    {
    case 1: PCmp = WP.PCmpWALTZR(sys); break;
    case 2: PCmp = WP.PCmpWALTZK(sys); break;
    case 3: PCmp = WP.PCmpWALTZQ(sys); break;
    case 4: PCyc = WP.CycWALTZ4(sys); break;
    case 5: PCyc = WP.CycWALTZ8(sys); break;
    case 6: PCyc = WP.CycWALTZ16(sys); break;
    }
//                      Set Acquistion and Profile Parameters
String IsoD = sys.symbol(0);                          // Detection/pulse channel
  String IsoG = WP.channel();                         // Decoupler channel
  if(IsoD == IsoG)                                    // Try and set channel to
    IsoD = sys.symbol(1);                             // not be the decoupling one
  double SW;                                          // Spectral width
  query_parameter(argc, argv, qn++,                   // Get desired spectral width
           "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                    // Block size (must be base 2)
  query_parameter(argc, argv, qn++,                   // Get block size
                 "\n\tBlock Size? ", npts);
  double lwhh = 3.0;                                  // Half-height linewidth
  query_parameter(argc, argv, qn++,                   // Ask for apodization strength
            "\n\tApodization (Hz)? ", lwhh);
  int NO = 30;                                        // # Of Offsets (on each side)
  query_parameter(argc, argv, qn++,                   // Get # offsets
   "\n\tNumber of Positive Decoupler Offsets? ", NO);
  double offset;
  query_parameter(argc, argv, qn++,                   // Get # offsets
   "\n\tDecoupler Offset Per Step (Hz)? ", offset);
//                  Set Up Variables Consistent Through All Offsets
double R = (lwhh/2)*HZ2RAD;                           // Set apodization rate
  gen_op Det = Fm(sys, IsoD);                         // Set detection operator to F-
  gen_op sigma0 = sigma_eq(sys);                      // Set density mx equilibrium
  gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);       // This is 90 detection pulse
  row_vector data(npts);                              // Block for acquisiton
/                   Set Up Variables Global Over Full Profile
  row_vector profile((2*NO+1)*npts, complex0);        // Block for profile
  double totaloff = double(NO)*offset;                // Total offset at end
  row_vector fidap;                                   // Block for apodized FID
  if(wt <=3) SW = PCmp.FIDsync(SW);                   // Synchronize dwell times
  else       SW = PCyc.FIDsync(SW);
  cout << "\n\tSynch Spectral Width " << SW;
  double td = 1/SW;                                   // Set dwell time
  double tt = (npts-1)*td;                            // Total FID length
```

```
 row_vector exp=Exponential(npts,tt,0.0,R,0);       // Block for apodization
//                              Loop Over Offsets, Calculate Profile
 int K =0;                                          // Point index in profile
 sys.offsetShifts(-NO*offset, IsoG);                // Set 1st profile offset
 cout << "\n\tProfile Offset ";
 for(int ov=-NO; ov<=NO; ov++)                      // Loop over offsets
   {
   printIndx(cout, ov);                             // Output offset index
   switch(wt)
     {
     case 1: PCmp = WP.PCmpWALTZR(sys); break;
     case 2: PCmp = WP.PCmpWALTZK(sys); break;
     case 3: PCmp = WP.PCmpWALTZQ(sys); break;
     case 4: PCyc = WP.CycWALTZ4(sys); break;
     case 5: PCyc = WP.CycWALTZ8(sys); break;
     case 6: PCyc = WP.CycWALTZ16(sys); break;
     }
   if(wt<=3) data=PCmp.FIDR(npts,td,Det,sigmap);// Perform acquisition
   else      data=PCyc.FID(npts,td,Det,sigmap);
   fidap = product(data,exp);                       // Apodized FID this offset
   data  = FFT(fidap);                              // Spectrum this offset
   profile.put_block(0, K, data);                   // Put spectrum in profile
   sys.offsetShifts(offset, IsoG);                  // Move system to next offset
   K += npts;                                       // Adjust profile point index
   }

 double F = totaloff + SW/2;                        // Final plot frequency
 GP_1D("prof.asc", profile, 0, -F, F);              // Output profile ASCII data
 GP_1Dplot("prof.gnu", "prof.asc");                 // Plot to screen using Gnuplot
 FM_1D("prof.mif", profile ,14,14,-F, F);           // Plot in FrameMaker MIF
 }
```

It is evident that use of the WALTZ pulse cycles is superior to use of repeated composite pulses without the compensating phase cycle (i.e. WALTZ-{4,8,16} works better than WALTZ-{R,K,Q}). It also is clear that, at least for this two spin system without relaxation effects, the WALTZ-16 is the best of these WALTZ sequences.

Unfortunately the source code for this example was well over a page long, mostly due to allowing the user to choose between decoupling schemes. Part of the length is also due to GAMMA use of Composite Pulses (WALTZ-{R,K,Q}) versus Pulse Cycles (WALTZ-{4,8,16}).

This difference is set in GAMMA so that the two types are internally handled in an efficient manner in simulations. In principle composite pulses could simply be pulse cycles with only 1 phase. Similarly, pulse cycles could be composite pulse where the number of steps would be multiplied by the number of phase changes the waveform is put through during the cycle.

Both of these will likely be less efficient in calculating decoupling effects! This is because GAMMA internally reuses Hamiltonians and propagators when possible. However, I will likely add in the ability to generate a WALTZ-K pulse cycle and a WALTZ-16 composite pulse since it will simmplify some GAMMA programs (at the expense of computation efficiency).

7 GARP

7.1 Overview

The module ***PulGARP***, which contains the class ***GARP***, facilitates the use of GARP pulse cycles in GAMMA nmr simulations. Class GARP contains parameters which define how GARP cycles is to be implemented and provides functions for building GARP based waveforms, composite pulses, and pulse trains.

7.2 Chapter Contents

7.2.1 GARP Section Listing

7.2.2 GARP Function Listing

Input/Output Functions

7.3 Constructors and Assignment

7.3.1 GARP

Usage:

```
#include <PulGARP.h>
GARP()
GARP(double gB1, const String& ch, double ph=0, double off=0);
GARP(const GARP& GRP)
```

Description:

The function *GARP* is used to create a GARP parameter container.

1. PulGARP() - Creates an “empty” NULL GARP parameter. Can be later filled by an assignment.
2. PulGARP(double gB1, const& ch, double ph, double off) - Sets up GARP for having an rf-field strength of ***gB1*** Hz on the channel specified by ***ch***. GARP will be applied with an overall phase of ***ph*** degrees and an offset of ***off*** Hz. Called with another PulGARP quantity this function constructs an identical PulGARP to the inputPWF1.
3. PulGARP(const PulGARP &PWF1) - Called with another PulGARP quantity this function constructs an identical PulGARP to the inputPWF1.

Return Value:

GARP returns no parameters. It is used strictly to create a GARP parameter container.

Examples:

```
PulGARP PG;
PulGARP PG1(538.9, “13C”);
PulGARP PG3(PG1);
```

See Also: =

7.3.2 =

Usage:

```
#include <PulGARP.h>
void GARP operator = (PulGARP &PWF1)
```

Description:

The unary operator = (the assignment operator) allows for the setting of one GARP to another GARP. If the GARP being assigned to exists it will be overwritten by the assigned GARP.

Return Value:

None, the function is void

Example:

```
PulGARP PG1(538.9, “13C”);
PulGARP PG3 = PG1;
```

See Also: PulGARP

7.4 Access Functions

7.4.1 channel

Usage:

```
#include <PulGARP.h>
String GARP::channel( )
```

Description:

The function ***channel*** returns a string indicating the isotope channel GARP is applied on.

Return Value:

The function returns a string.

Example:

```
#include <PulGARP.h>
GARP GP(600.0, “13C”);                    // GARP Parameters
cout << “\n\tGARP Decouple On “           // Output channel
     << GP.channel();
```

See Also:

7.4.2 strength

Usage:

```
#include <PulGARP.h>
double GARP::strength()
```

Description:

The function ***strength*** returns the value of the rf-field amplitude used in GARP (in Hz).

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
GARP GP(600.0, “13C”);                    // GARP Parameters
cout << “\n\tGARP Field Strength Is “     // Output rf strength
     << GP.strength() << “ Hz”;           // (will be 600 Hz of course)
```

See Also:

7.4.3 phase

Usage:

```
#include <PulGARP.h>
double GARP::phase()
```

Description:

The function ***phase*** returns the value of the rf-field phase used for GARP in degrees.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
GARP GP;                                    // Declare GARP Parameters
GP.read(“filein.pset”)                      // Read in GARP Parameters
cout << “\n\tGARP Phase Is “                // Output (overall) rf phase
     << GP.phase() << “ degrees”;
```

7.4.4 offset

Usage:

```
#include <PulGARP.h>
double GARP::offset()
```

Description:

The function ***offset*** returns the value of the rf-field offset used for GARP in Hz.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
GARP GP;                                    // Declare GARP Parameters
GP.read(“filein.pset”)                      // Read in GARP Parameters
cout << “\n\tGARP Offset Is “               // Output rf offset
     << GP.offset() << “ Hz”;
```

7.5 GARP-1 Functions

7.5.1 WF

7.5.2 WF_GARP1

Usage:

```
#include <PulGARP.h>
PulWaveform GARP::WF()
PulWaveform GARP::WF_GARP1()
```

Description:

The ***GARP*** member functions ***WF*** and WF_GARP1 both return the 25 step GARP-1 waveform.

Return Value:

A 25 step GARP-1 pulse waveform is returned.

Example:

```
#include <PulGARP.h>
GARP GP;                          // Declare GARP Parameters
GP.read("filein.pset")            // Read in GARP Parameters
PulWaveform GWF = GP.WF();        // Make GARP waveform (GARP-1)
```

See Also: PCmp, CycGARP1

7.5.3 PCmp

7.5.4 PCmpGARP1

Usage:

```
#include <PulGARP.h>
PulComposite GARP::PCmp(const spin_system& sys)
PulComposite GARP::PCmpGARP1(const spin_system& sys)
```

Description:

The ***GARP*** member functions ***PCmp*** and ***PCmpGARP1*** both return the 25 step GARP-1 composite pulse for the input spin system ***sys***.

Return Value:

A 25 step GARP-1 composite pulse is returned.

Example:

```
spin_system sys;                  // Declare a spin system
sys.read("filein.sys");           // Read in the spin system
```

```
GARP GP;                              // Declare GARP Parameters
GP.read("filein.pset")                // Read in GARP Parameters
PulComposite GCP = GP.PCmp(sys);      // GARP composite pulse(GARP-1)
```

See Also: WF, CycGARP1

7.5.5 CycGARP1

Usage:

```
#include <PulGARP.h>
PulCycle GARP::CycGARP1(const spin_system& sys)
```

Description:

The ***GARP*** member function ***CycGARP1*** returns a pulse cycle using the 25 step GARP-1 pulse sequence coupled to a WALTZ-4 cycle.

Return Value:

A GARP-1 pulse cycle is returned.

Example:

```
spin_system sys;                      // Declare a spin system
sys.read("filein.sys");               // Read in the spin system
GARP GP;                              // Declare GARP Parameters
GP.read("filein.pset")                // Read in GARP Parameters
PulCycle GCy = GP.CycGARP1(sys);      // GARP-1 pulse cycle
```

See Also: WF, PCmp

7.6 Propagator Functions

7.6.1 GetU

Usage:

```
#include <PulGARP.h>
gen_op GetU(i)
```

Description:

The function ***GetU*** will return the propagator which is active during step ***i*** of the pulse waveform. The Hamiltonian returned is defined in the rotating frame of the pulse waveform and contains contributions from the pulse waveform rf-field and the isotropic static Hamiltonian.

Return Value:

The function returns an operator.

Example:

```
#include <PulGARP.h>
double tp = 0.01;                    // Set pulse length to 10 ms
int N = 1001;                        // Set number of steps to 1001
```

See Also:

7.7 Input/Output Functions

7.7.1 printBase

Usage:

```
#include <PulGARP.h>
ostr printBase(ostream& ostr)
```

Description:

The function ***printBase*** will put information regarding the GARP parameters into the output stream ***ostr*** given as an input argument. The function will have less embellishment than the similar function ***print***.

Return Value:

The function modifies the output stream and returns it.

Example:

```
GARP GP;
GP.read("filein.pset");
GP.print(cout);
```

See Also: print, <<

7.7.2 print

Usage:

```
#include <PulGARP.h>
ostr print(ostream& ostr)
```

Description:

The function ***print*** will put information regarding the GARP parameters into the output stream ***ostr*** given as an input argument.

Return Value:

The function modifies the output stream and returns it.

Example:

```
GARP GP;
GP.read("filein.pset");
GP.print(cout);
```

See Also: printBase, <<

7.7.3 <<

Usage:

```
#include <PulGARP.h>
ostream& operator << (ostream& ostr, PulGARP& PG)
```

Description:

The operator **<<** adds the GARP parameters specified as an argument ***PG*** to the output stream ***ostr***.

Return Value:

None.

Example:

```
Garp GP;
GP.read("filein.pset");
cout << GP;
```

See Also: print, printBase

7.8 Auxiliary Functions

7.8.1 channel

Usage:

```
#include <PulGARP.h>
String PulGARP::channel( )
```

Description:

The function ***channel*** returns a string indicating the isotope channel the pulse waveform is applied on.

Return Value:

The function returns a string.

Example:

```
#include <PulGARP.h>
PulGARP PW = GARP(600.0, “13C”);                    // GARP-1 Waveform
cout << “\n\tGARP Decouple On “ << PW.channel();// Output channel
```

See Also:

7.8.2 steps

Usage:

```
#include <PulGARP.h>
int PulGARP::steps( )
```

Description:

The function ***steps*** returns the number of individual steps defined in the pulse waveform.

Return Value:

The function returns an integer.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.3 cycles

Usage:

```
#include <PulGARP.h>
int PulGARP::cycles( )
```

Description:

The function ***cycles*** returns the number of individual cycle steps defined in the pulse waveform.

Return Value:

The function returns an integer.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.4 name

Usage:

```
#include <PulGARP.h>
String PulGARP::name( )
```

Description:

The function ***name*** returns the name of the pulse waveform.

Return Value:

The function returns a string.

Example:

7.8.5 values

Usage:

```
#include <PulGARP.h>
row_vector PulGARP::values( )
```

Description:

The function ***values*** returns a row_vector containing values which define the pulse waveform steps. The ith vector value contains the values $\{\gamma B1, \phi\}$, where the real component is the rf-field strength in Hz, and the imaginary component is the rf-phase in degrees (or radians).

Return Value:

The function returns a row vector.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.6 value

Usage:

```
#include <PulGARP.h>
complex PulGARP::value(int i)
```

Description:

The function ***value*** returns a compex number for the values which define the pulse waveform step ***i***. The value contains the number {γB1, ϕ}, where the real component is the rf-field strength in Hz, and the imaginary component is the rf-phase in degrees (or radians).

Return Value:

The function returns a complex number.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.7 phase

Usage:

```
#include <PulGARP.h>
double PulGARP::phase(int i)
```

Description:

The function ***phase*** returns the value of the rf-field phase at pulse waveform step ***i*** in degrees (or radians).

Return Value:

The function returns a double.

7.8.8 strength

Usage:

```
#include <PulGARP.h>
double PulGARP::strength(int i)
```

Description:

The function ***strength*** returns the value of the rf-field amplitude at pulse waveform step ***i*** in Hz.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.9 length

Usage:

```
#include <PulGARP.h>
double PulGARP::length(int i)
```

Description:

The function ***length*** returns the length of th pulse waveform in seconds.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.10 steplength

Usage:

```
#include <PulGARP.h>
double PulGARP::steplength(int i)
```

Description:

The function ***steplength*** returns the length of an individual pulse waveform step in seconds.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
```

7.8.11 cyclelength

Usage:

```
#include <PulGARP.h>
double PulGARP::cyclelength(int i)
```

Description:

The function ***cyclelength*** returns the length of the pulse waveform cycle in seconds.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.12 scyclelength

Usage:

```
#include <PulGARP.h>
double PulGARP::scyclelength(int i)
```

Description:

The function ***scyclelength*** returns the length of th pulse waveform supercycle in seconds.

Return Value:

The function returns a double.

Example:

```
#include <PulGARP.h>
```

See Also:

7.8.13 FZ

Usage:

```
#include <PulGARP.h>
int PulGARP::FZ( )
```

Description:

The function ***FZ*** returns the z-axis spin operator associated with the pulse waveform. This operator will be selective for the isotope which the pulse waveform affects.

Return Value:

The function returns an operator.

Example:

```
#include <PulGARP.h>
```

See Also:

7.9 Description

7.9.1 Introduction

The functions in module ***PulGARP*** and Class ***GARP*** (contained in module ***PulGARP***), is designed to facilitate the use of GARP[1] pulse trains in GAMMA NMR simulation programs. In GAMMA, as in an NMR experiment, we should like to use GARP pulse trains as individual steps in a general pulse sequence, including use in variable delays as part of multi-dimensional experiments and/or use in pulse trains during acquisition steps.

7.9.2 GARP Parameters

A variable of type GARP contains only primitive parameters: e consider a pulse waveform as involving four basic features: 1.) A ***# steps***, 2.) An ***rf-field strength***, 3.) An ***rf-phase*** 4.) An ***rf-offset***. These value can be used to completely determine how to set up composite pulses such as that used in a GARP-1 pulse train.

7.9.3 Basic GARP Waveform

GARP sequences are based on a 25 step composite pulse. The pulses are applied with the same rf-strength but vary in their applied length and oscillate phase between ϕ and $\phi+\pi$. The details are shown in the following figure.

Basic GARP 25 Step Sequence

Step	Angle	Step	Angle	Step	Angle
1	30.5	9	134.5	17	258.4
2	$\overline{55.2}$	10	$\overline{256.1}$	18	$\overline{64.9}$
3	257.8	11	66.4	19	70.9
4	$\overline{268.3}$	12	$\overline{45.9}$	20	$\overline{77.2}$
5	69.3	13	25.5	21	98.2
6	$\overline{62.2}$	14	$\overline{72.7}$	22	$\overline{133.6}$
7	85.0	15	119.5	23	255.9
8	$\overline{91.8}$	16	$\overline{138.2}$	24	$\overline{65.6}$
				25	$\overline{53.4}$

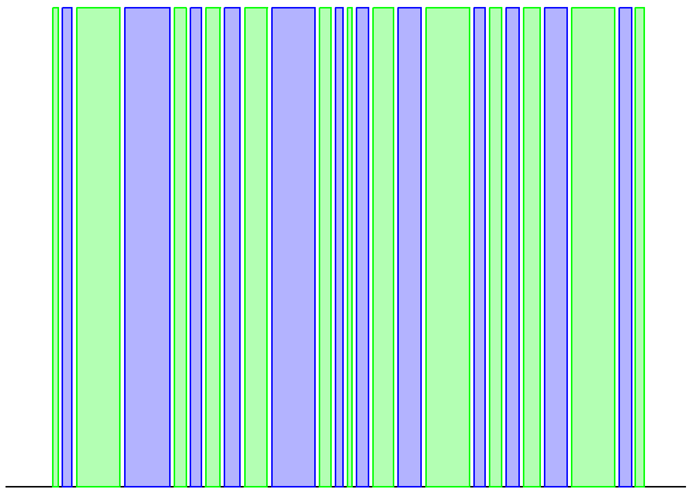


Figure 7-1 The basic 25-step GARP waveform. The blue steps indicate pulse that are applied with a 180 degree phase shift, as indicate by a bar in the table listing. The program which produced this plot can be found at the end of this chapter, GarpWF0.cc on page 130.

1. For information on GARP see the article by Shaka, Barker and Freeman in *J. Magn. Reson.*, **64**, 547-552 (85). GARP = Globally optimized Alternating-phase Rectangular Pulses.

7.9.4 GARP-1 Pulse Cycle

The GARP-1 decoupling sequence repeatedly uses the GARP 25 step composite pulse but changes the overall phase in a WALTZ-4 . Thus for GARP-1 we have

GARP-1 Decoupling Sequence

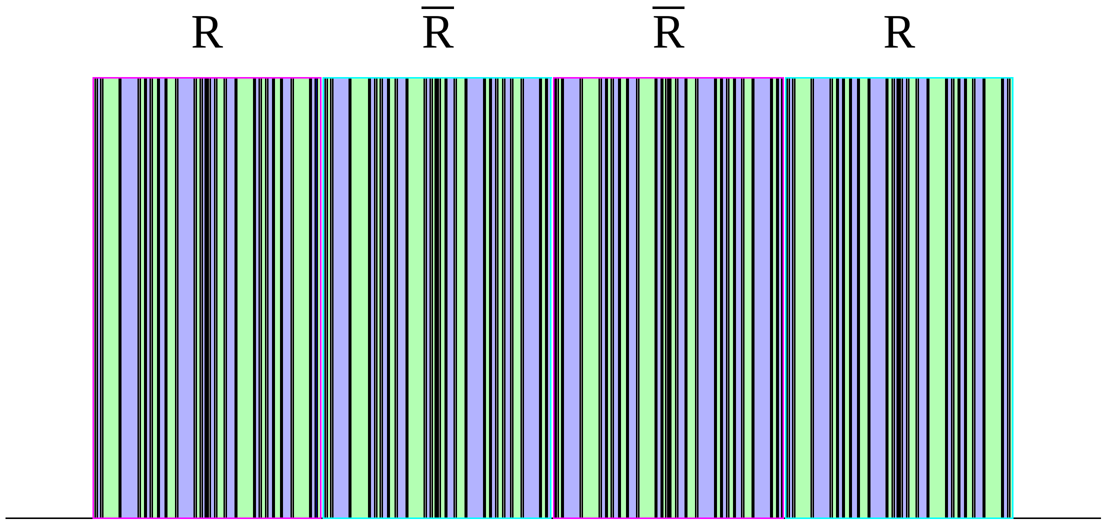


Figure 7-1 The GARP-1 decoupling sequence. Each 25-step GARP waveform is repeated with the 4-step phase adjustment overlaid: 0, 180, 180, 0. These are designated R, $\overline{R}$, $\overline{R}$, R respectively. The 4 waveforms (100 steps) of GARP-1 are repeated as long as the decoupling sequence is applied.

The GARP 25-step waveforms (composite pulses) are continuously applied with the same rf-strength but will change phase between ϕ and $\phi+\pi$. As is typical in such sequence, the cycle phases are changed in a 4-step sequence: 0, 180, 180, 0. Thus the first and last 25 steps of the GARP-1 cycle will be identical as will the second and third 25 steps. But these two types are 180 degrees out of phase.

7.10 GARP Parameters

This section describes how an ASCII file may be constructed that is self readable by a GARP variable. The file can be created with an editor of the users choosing and is read with the GARP member function “read”. This provides for an extremely flexible and program independent means of implementing GARP in NMR simulations.

The GARP (ASCII) input file is scanned for the specific parameters which specify the pulse-delay parameters[1]: delay length, rf-length, rf-strength, rf-phase, rf-offset, pulse angle, and the number of GARP step. These parameters are recognized by certain keywords, as shown in the following table.

Table 2: Spin System Parameters

Parameter Keyword	Assumed Units	Examples Parameter (Type) : Value - Statement
GARPgamB1	Hz	GARPgamB1 (1) : 600.0 - Field Strength (Hz)
GARPiso	none	GARPiso (2) : 19F - GARP rf pulse channel
GARPphi	degrees	GARPphi (1) : 2.0 - GARP rf phase (deg)
GARPstps	none	GARPstps (0) : 20 - GARP pulse-delay steps

The order in which these parameters reside in the ASCII file is of no consequence.

The format of each parameter is quite simple and general for all GAMMA parameters. At the beginning of a line the parameter keyword is written followed by an optional index number in parenthesis. This is then followed by one or more blanks and then an integer in parentheses. The integer corresponds to the type of parameter value: 0 = integer, 1 = floating point, or 2 = string. Following the parenthesis should be at least one blank then a colon to indicate the parameter value follows. The parameter value is then written followed by some blanks then a hyphen followed by an optional comment.

There is one major restriction; keywords and string parameters cannot contain blanks. For example, v (0) is unknown, v(0) is. The string value 19 F is unknown, 19F is fine. If multiple GARP pulse-delay steps need to be defined in the same file then simply put an index on all parameters associated with a desired GARP and read the parameters using that index.

To read the file, see the documentation for function read (or ask-read). There is also an example program readsystem.cc provide at the end of this Chapter which should indicate how the file is read. Each of the possible spin system input parameters is now described in more detail.

1. Note that the ASCII file must contain viable parameters in GAMMA format. Indeed, the file is a GAMMA parameter set and, as such, may contain any amount of additional information along with the valid GARP parameters.

Channel: GARPiso

This parameter is optional. It will define which isotope channel the GARP rf-pulse will be applied on. If no channel is specified GAMMA will assume that all spins in the system being treated are affected by the rf. Thus if no channel is specified and GARP is utilized in an NMR simulation the system should be homo-nuclear or the same GARP should be desired on all channels (same offset, phase, etc.)

Channel: GARPphi

This parameter is optional. It will define the rf-phase of the GARP pulse. If no phase is specified it will be taken to be zero.

Pulse Length: GARPtp

The parameters { GARPang, GARPgamB1, GARPtp } work together. GARPtp will set the pulse length if either GARPang and/or GARPgamB1 have also been specified. If only GARPtp has been specified amongst the three an error will result when reading these parameters to define GARP.

Pulse Strength: GARPgamB1

The parameters { GARPang, GARPgamB1, GARPtp } work together. GARPgamB1 se the pulse strength if either GARPang or GARPgamB1 have also been specified. If only GARPgamB1 is specified amongst the three an error will result when reading these parameters to define GARP. If all three parameters have been specified then GARPgamB1 will be ignored, the strength set by { GARPang, GARPtp }

Sync Frequency: GARPF

This parameter sets the GARP frequency, i.e. the pulse-delay repetition rate. Thus, users may specify the specific frequency that GARP will affect the strongest. The parameter will override any delay time set by the parameter GARPtp. The combined length of the GARP pulse and delay will be set to 1/GARPF.

Delay Length: GARPtp

This parameter sets the GARP delay length, independent of the GARP pulse length. If GARPF exists then this parameter will not be used.

7.11 GARP Examples

7.11.1 Reading GARP Parameters

To keep GAMMA programs using GARP sequences versatile, users will want to keep all GARP specifications undetermined in the code. As the program runs, GARP settings are either specified interactively and/or read in from an external ASCII (parameter) file. This section gives examples of the latter case. The figure below shows an ASCII parameter file on the left and some GAMMA program code on the right.

Reading GARP Parameters

```
GARPphi        (1) : 90.0    - GARP overall rf-phase (deg)
GARPiso        (2) : 1H      - GARP pulse channel
GARPgamB1      (1) : 983.0   - GARP pulse strength (Hz)
GARPiso(3)     (2) : 19F     - GARP pulse channel
GARPgamB1(3)(2) : 2045.9     - GARP pulse strength (Hz)
```

```
GARP GP;                           // Declare GARP parameters
GP.read("GARP.pset");              // Read GARP from file
GP.read("GARP.pset", 3);           // Read GARP from file
GP.ask_read(argc, argv, 1);        // Read GARP from file
GP.ask_read(argc, argv, 2, 3);     // Read GARP from file
```

Figure 7-2 The basic 25-step GARP waveform. The blue steps indicate pulse that are applied with a 180 degree phase shift, as indicate by a bar in the table listing. The program which produced this plot can be found at the end of this chapter, GarpWF0.cc on page 130.

The ASCII parameter file (on the left) is taken to be called “GARP.pset” and is read in by the program code. Thus one can change the GARP parameters independent of the GAMMA code. The ASCII file format is typical of GAMMA parameter sets: The line ordering is of no consequence, the column spacing is not important, the end “- comments” can be left off, and additional lines of text or parameters may be included.

The GAMMA code is color coded with the parameters they read in the previous figure. Thus the second line (blue) will read the blue parameters and set up GARP with a strength of 983 Hz on the proton channel with an overall phase of 90 degrees. Similarly, the next line (green) will read the parameters colored green from the same ASCII file but sets up GARP with a strength of 2.0459 kHz on the 19F channel (no phase, no offset).

The next line will interactively ask the user to supply a filename where the program can get to GARP parameters. This filename (in this case “GARP.pset”) will be prompted for unless the user specifies the file on the command line when the program is executed. The following line does the same but reads the GARP parameters indexed with a “3” from the file.

Using a combination of these commands, the user has complete flexibility in defining one or more GARP sequences in the same GAMMA program. The GARP parameters can be easily changed by either changing their values in the ASCII file and/or changing the filename given to the program. See the section of GARP parameters to see which parameters can be used in setting up GARP sequences. See the other programs in this chapter for full examples GAMMA programs using them.

7.11.2 GARP-1 Decoupling

In this section we shall produce a simple 1D NMR spectrum under GARP-1 decoupling. A hard 90 pulse will be applied to a chosen spin system on the acquisition channel. Then GARP-1 will be applied on the decoupler channel during acquisition. The resulting FID will be apodized and Fourier transformed, the NMR spectrum put on screen using Gnuplot. Note that relaxation and exchange effects will be ignored in this simulation. The code for simple GARP-1 decoupling is shown below:

```
/* GARPdec0.cc **********************************************************************
**                                                                                  **
**                     GAMMA Decoupling Example Program                             **
**                                                                                  **
**  This program uses the class GARP to perform a simple decoupling                 **
**  simulation.  A hard ideal pulse will be applied to an input spin                **
**  system.  Subsequently, an acquisition will be performed with GARP-1             **
**  decoupling applied on a specified channel.                                      **
**                                                                                  **
** Author:      S.A. Smith                                                          **
** Date:        3/9/98                                                              **
** Update:      3/9/98                                                              **
** Version:     3.5.4                                                               **
** Copyright:   S. Smith.  Modify this program as you see fit for personal          **
**              use, but you must leave the program intact if redistrubuted         **
**                                                                                  **
*************************************************************************************
**/

#include <gamma.h>
#
main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tGARP Decoupling\n\n";
  int qn = 1;                                        // Query index
  spin_system sys;                                   // Declare a spin system
  GARP GP;                                           // GARP parameters
  String filename = sys.ask_read(argc,argv,qn++);    // Ask for/read in the system
  cout << sys;                                       // Have a look (for setting SW)
  GP.read(filename);                                 // Read in GARP parameters
  PulCycle PCyc = GP.CycGARP1(sys);                  // Construct GARP-1 cycle
  String IsoD = sys.symbol(0);                       // Detection/pulse channel
  if(sys.heteronuclear())                            // If heteronuclear system
    query_parameter(argc, argv, qn++,                // ask for detection channel
          "\n\tDetection Isotope? ", IsoD);
  double SW;                                         // Spectral width
  query_parameter(argc, argv, qn++,                  // Get desired spectral width
          "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                   // Block size (must be base 2)
  query_parameter(argc, argv, qn++,                  // Get block size
             "\n\tBlock Size? ", npts);
  double lwhh = 1.0;                                 // Half-height linewidth
  query_parameter(argc, argv, qn++,                  // Ask for apodization strength
          "\n\tApodization (Hz)? ", lwhh);
  double td = 1/SW;                                  // Set dwell time
  double R = (lwhh/2)*HZ2RAD;                        // Set apodization rate
  double tt = (npts-1)*td;                           // Total FID length
  gen_op H = Ho(sys);                                // Set isotropic Hamiltonian
  gen_op Det = Fm(sys, IsoD);                        // Set detection operator to F-
  gen_op sigma0 = sigma_eq(sys);                     // Set density mx equilibrium
  gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);      // This is 1st 90 pulse
  row_vector data = PCyc.FID(npts,td,Det,sigmap);    // Perform acquisition under GARP-1
  row_vector exp = Exponential(npts,tt,0.0,R,0);     // Here is an exponential
  row_vector fidap = product(data,exp);              // Apodized FID
  data = FFT(fidap);                                 // Transformed FID -> spectrum
  GP_1D("spec.asc", data, 0, -SW/2, SW/2);           // Output ASCII file
  GP_1Dplot("spec.gnu", "spec.asc");                 // Plot to screen using Gnuplot
  }
```

The first half of this program simply sets the parameters up interactively. Note that both the spin system and the GARP parameters are contained in the same file who’s name the user must specify.

The second half of the program does the simulation. The FID is acquired using the pulse cycle function “FID” and the pulse cycle has been set to GARP-1. The last few lines apodize and transform the FID then spit the plot out on the screen.

Addition of relaxation & exchange effects and/or changing the 1st pulse to non-ideal will require only minor modifications of a few lines.

The input parameter (ASCII) file is listed on the following page. It contains parameters for both the spin system and the GARP settings.

Example of a GARP decoupling input file (GARPdec.sys)

```
SysName     (2) : GARP      - Name of the Spin System
NSpins      (0) : 4         - Number of Spins in the System
Iso(0)      (2) : 1H        - Spin Isotope Type
Iso(1)      (2) : 1H        - Spin Isotope Type
Iso(2)      (2) : 1H        - Spin Isotope Type
Iso(3)      (2) : 13C       - Spin Isotope Type
v(0)        (1) : 105.0     - Chemical Shifts in Hz
v(1)        (1) : -174.32   - Chemical Shifts in Hz
v(2)        (1) : 15.0      - Chemical Shifts in Hz
v(3)        (1) : 0.0       - Chemical Shifts in Hz
J(0,1)      (1) : 10.0      - Coupling Constants in Hz
J(0,2)      (1) : 7.9       - Coupling Constants in Hz
J(0,3)      (1) : 22.0      - Coupling Constants in Hz
J(1,2)      (1) : 2.8       - Coupling Constants in Hz
J(1,3)      (1) : 32.0       - Coupling Constants in Hz
J(2,3)      (1) : 18.3      - Coupling Constants in Hz
Omega       (1) : 400       - Spect. Freq. in MHz (1H based)

GARPphi     (1) : 0         - GARP pulse phase (deg)
GARPiso     (2) : 13C       - GARP pulse channel
GARPgamB1   (1) : 1500.0    - GARP pulse strength (Hz)
```

When the program (GARPdec0.cc) is compiled its execution will produce a plot on screen if the Gnuplot program is available. Assuming the executable is called a.out, the following command will produces a spectrum:

```
a.out GARPdec.sys 1H 500 1024 .5
```

The command “a.out” alone will prompt you for input values. Had you input the above command (or parameters) the spectrum should appear as shown in the following figure.

^{13}C Decoupled Spectrum Using GARP-1

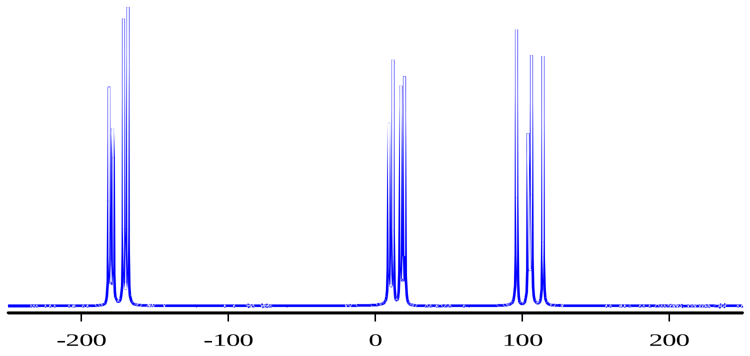


Figure 7-3 The spectrum produced using the program GARPdec0.cc with input parameter file GARPdec.sys. The decoupler was applied to the 13C channel with a 1.5 kHz field strength. Detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed with a 0.5 Hz line-broadening.

By either editing the input file or specifying a different input file, the spectrum can be radically altered. For example, by setting the GARP pulse strength to zero one obtains the following spectrum.

^{13}C Coupled Spectrum, Zero Strength GARP-1

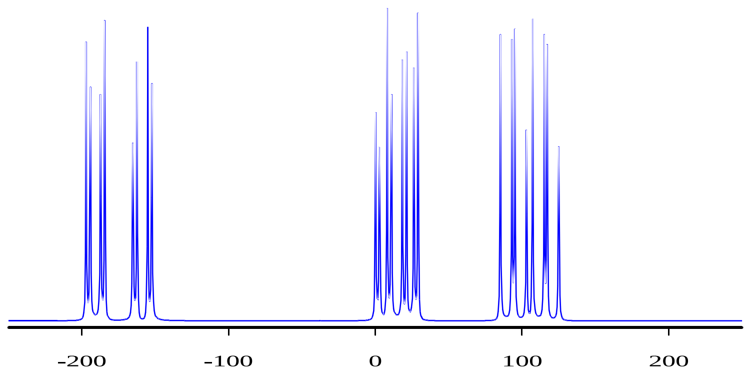


Figure 7-4 Same as previous figure but with no decoupler field strength.

7.11.3 GARP-1 Decoupling vs. Field

We can readily modify the previous program to loop over differing rf-field strengths and determine how well GARP-1 does as decoupling. In this case we will just read in a series of gB1 values from an external ASCII file and loop over them producing a 1D spectrum at each value. We'll spit out all the spectra in a single stack plot.

```
/* GARPdec1.cc ************************************************************************
**                                                                                   **
**                         GAMMA Decoupling Test Program                             **
**                                                                                   **
**  This program uses the class GARP to perform a simple decoupling                  **
**  simulation.  A hard ideal pulse will be applied to an input spin                 **
**  system.  Subsequently, an acquisition will be performed with GARP-1              **
**  decoupling applied on a specified channel.                                       **
**                                                                                   **
** Author:      S.A. Smith                                                           **
** Date:        3/11/98                                                              **
** Update:      3/11/98                                                              **
** Version:     3.5.4                                                                **
** Copyright:   S. Smith.  You can modify this program as you see fit                **
**              for personal use, but you must leave the program intact              **
**              if you re-distribute it.                                             **
**                                                                                   **
*************************************************************************************/

#include <gamma.h>

main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tGARP Decoupling Vs. Decoupler Strength\n\n";
  int qn = 1;                                     // Query index
  spin_system sys;                                // Declare a spin system
  GARP GP;                                        // GARP parameters
  String filename;                                // Input filename
  filename = sys.ask_read(argc,argv,qn++);        // Ask for/read in the system
  cout << sys;                                    // Have a look (for setting SW)
  GP.read(filename);                              // Read in GARP parameters
  cout << GP;
  PulCycle PCyc;

  query_parameter(argc, argv, qn++,               // Ask for field strength file
     "\n\tFile of Field Strengths? ", filename);
  int N;                                          // Number of field strengths
  double* gB1s = GetDoubles(filename, N);         // Get array of field strengths
  String IsoD = sys.symbol(0);                    // Detection/pulse channel
  if(sys.heteronuclear())                         // If heteronuclear system
    query_parameter(argc, argv, qn++,             // ask for detection channel
         "\n\tDetection Isotope? ", IsoD);
  double SW;                                      // Spectral width
  query_parameter(argc, argv, qn++,               // Get desired spectral width
         "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                // Block size (must be base 2)
  query_parameter(argc, argv, qn++,               // Get block size
             "\n\tBlock Size? ", npts);
  double lwhh = 3.0;                              // Half-height linewidth
  query_parameter(argc, argv, qn++,               // Ask for apodization strength
         "\n\tApodization (Hz)? ", lwhh);
  double td = 1/SW;                               // Set dwell time
  double R = (lwhh/2)*HZ2RAD;                     // Set apodization rate
  double tt = (npts-1)*td;                        // Total FID length
  gen_op H = Ho(sys);                             // Set isotropic Hamiltonian
  gen_op Det = Fm(sys, IsoD);                     // Set detection operator to F-
  gen_op sigma0 = sigma_eq(sys);                  // Set density mx equilibrium
  gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);   // This is 1st 90 pulse
  row_vector data, exp, fidap;
  matrix datamx(N,npts);
  for(int i=0; i<N; i++)
    {
    GP.strength(gB1s[i]);                         // Set GARP field strength
    PCyc = GP.CycGARP1(sys);                      // Set GARP-1 pulse cycle
    data = PCyc.FID(npts,td,Det,sigmap);          // Perform acquisition
    exp = Exponential(npts,tt,0.0,R,0);           // Here is an exponential
    fidap = product(data,exp);                    // Apodized FID
    data = FFT(fidap);                            // Transformed FID -> spectrum
    datamx.put_block(i,0,data);
    }
  double Nm1 = double(N-1);
  String AF("stk.asc");
  String GF("stk.gnu");
  GP_stack(AF, datamx, 0,1,N,0.0,Nm1);
  GP_stackplot(GF, AF);
  FM_stack("stk.mif", datamx, 1.5, 1.5, 1);
  }
```

The modifications from the previous program are obvious. An external ASCII file is used to specify a list of decoupler field strengths and these are read into the program. These fields are looped over, a new spectrum computed at each decoupler strength. The spectra are put into a matrix which is given to the Gnuplot routines for display as a stack plot on screen. In addition, the stack plot is output in FrameMaker MIF format for incorporation into documents in an editable form. The latter is shown in the following figure.

^{13}C GARP-1 Decoupling Versus RF-Field Strength

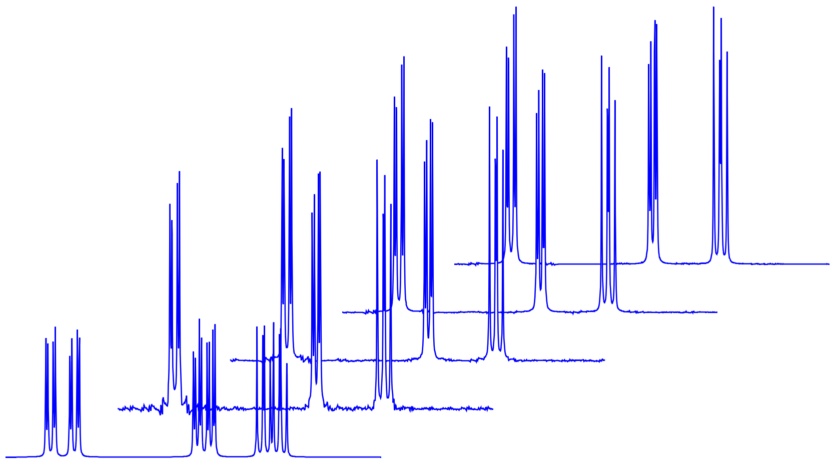


Figure 7-5 Proton spectra produced using the program GARPdec1.cc with input parameter file GARPdec.sys and decoupler strength file GARPdecBs. The decoupler was applied to the 13C channel with field strengths shown. Detection was on the proton channel. 1K data points were collected using a spectral width of 500 Hz. The data was processed with a 1.0 Hz line-broadening.

When the program (GARPdec1.cc) is compiled its execution will produce a stack plot on screen if the Gnuplot program is available. Assuming the executable is called a.out, the following command will produce the plot shown in the previous figure:

```
a.out GARPdec.sys GARPdecBs 1H 500 1024 1.0
```

The ASCII file GARPdecBs contains a list of rf-field strengths (in Hz) that the program used. The file has a single field strength per line and is shown next.

GARP decoupling rf-field input file (GARPdecBs)

```
0
200
400
600
800
```

Unlike GAMMA parameter set files (such as GARPdec.sys) this file is simple ASCII and cannot have anything other than a single floating point or integer value per line. No additional comments may be included.

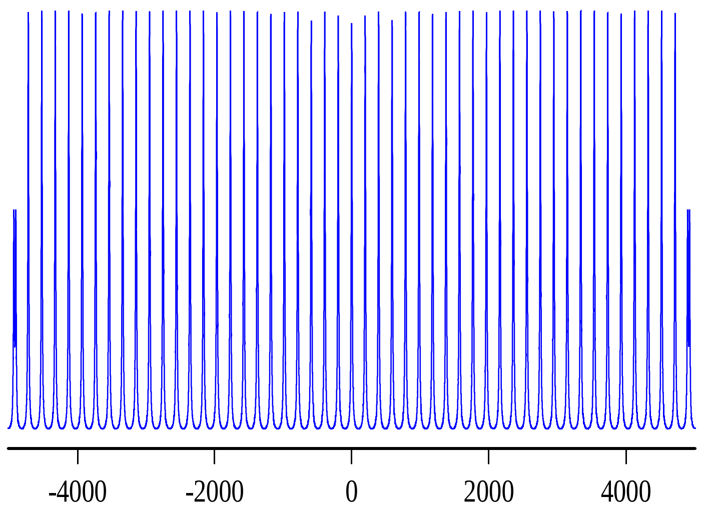
7.11.4 GARP-1 Decoupling Profile

In this section we shall attempt to produce a GARP-1 decoupling profile. A hard 90 pulse will be applied to a simple heteronuclear spin system on the acquisition channel. Then GARP-1 will be applied on the decoupler channel during acquisition. The resulting FID will be apodized and Fourier transformed. This pulse-delay process will be repeated for differing offsets on the decoupler channel. Each spectrum will be plotted with its center at the offset frequency to produce the profile.

The really no significant differences between this and our previous calculations. To determine a profile one uses the simplest spin system (here a two spin heteronuclear system). The 1D spectrum is recalculated after either moving the decoupler rf offset or, equivalently, moving all decoupler isotope channel chemical shifts. The spectra are all just put into a single vector, offset so their respective centers are set to be referenced to the decoupler offset value.

For fun, we'll design the GAMMA program to allow for the reproduction of Fig 1c. (bottom) in the original GARP-1 publication by Shaka, Barker, and Freeman (page 550). In fact, here are the GAMMA simulation results.

GARP-1 Decoupling Profile



```
SysName      (2) : GARP    - Name of the Spin System
NSpins       (0) : 2       - Number of Spins
Iso(0)       (2) : 1H      - Spin Isotope Type
Iso(1)       (2) : 13C     - Spin Isotope Type
v(0)         (1) : 0.0     - Chemical Shifts in Hz
v(1)         (1) : 0.0     - Chemical Shifts in Hz
J(0,1)       (1) : 221.0   - Coupling Constants in Hz
Omega        (1) : 720     - Spec. Freq. in MHz

GARPphi      (1) : 0       - GARP pulse phase (deg)
GARPiso      (2) : 13C     - GARP pulse channel
GARPgamB1    (1) : 2000.0  - GARP pulse strength (Hz)
```

Figure 7-6 GARP-1 decoupling profile. Decoupling was performed on the carbon channel in a ^{13}C-^{1}H two spin system. The decoupling rf-field strength was set to 2 kHz and the scalar coupling to 221 Hz. A linebroadening of 1.5 Hz was used in processing the spectra. The block size was 1K and the offset increment set to 200 Hz. These parameters were use to mimic the Shaka et. al paper. The text at the right is the file which was fed into the simulation program.

The agreement is excellent. The code for a "synchronous" GARP-1 decoupling profile is shown on the next page. This program sets a spectral width such that acquisition points are taken only after an even number of GARP-1 cycles (or at least an even number of GARP 25-step waveforms). In examining the program note that there are very few lines that have much to do with GARP. A quick replacement of a couple of lines would make this use MLEV or WALTZ or ..... Additionally we could adjust it let the user select among decoupling sequences. Even better, we can perform a slight adjustment and include the effects of relaxation and/or exchange.

```
/* GARPprof1.cc ***********************************************************************
**                                                                                  **
**                  GAMMA Decoupling Test Program                                   **
**                                                                                  **
** This program uses the class GARP to perform a simple decoupling                  **
** simulation.  A hard ideal pulse will be applied to a simple two spin  hetero-    **
** nuclear system.  Subsequently, an acquisition will be performed with             **
** GARP-1 decoupling applied on one the channel which is not being                  **
**  idetected.  This process will be repeated over a range of decoupler offsets.   **
** The result is a GARP-1 decoupler profile and will be plotted on screen if        **
** Gnuplot is available on the system.  The profile is also output inMIF.           **
**                                                                                  **
** Author:      S.A. Smith                                                          **
** Date:        3/9/98                                                              **
** Update:      3/9/98                                                              **
** Version:     3.5.4                                                               **
** Copyright:  S. Smith.  You can modify this program as you see fit for your       **
**                use, but you must leave the program intact if distrubuted.        **
**                                                                                  **
*************************************************************************************/

#include <gamma.h>

main(int argc, char* argv[])
  {
  cout << "\n\t\t\t\tGARP Decoupling Profile\n\n";
//             Read In Spin System & GARP Parameters
int qn = 1;                                          // Query index
  spin_system sys;                                   // Declare a spin system
  GARP GP;                                           // GARP parameters
  String filename;                                   // Input filename
  filename = sys.ask_read(argc,argv,qn++);           // Ask for/read in the system
  cout << sys;                                       // Have a look (for setting SW)
  if(sys.spins()!=2 || sys.homonuclear())
    cout << "\n\tWarning! This program has been"
         << " set up for a two spin heteronuclear"
         << " system.\n Results on other systems"
         << " can be unpredictable........";
  GP.read(filename);                                 // Read in GARP parameters
//             Set Acquistion and Profile Parameters
String IsoD = sys.symbol(0);                         // Detection/pulse channel
  String IsoG = GP.channel();                        // Decoupler channel
  if(IsoD == IsoG)                                   // Try and set channel to
    IsoD = sys.symbol(1);                            // not be the decoupling one
  double SW;                                         // Spectral width
  query_parameter(argc, argv, qn++,                  // Get desired spectral width
             "\n\tSpectral Width (Hz)? ", SW);
  int npts = 1024;                                   // Block size (must be base 2)
  query_parameter(argc, argv, qn++,                  // Get block size
                   "\n\tBlock Size? ", npts);
  double lwhh = 3.0;                                 // Half-height linewidth
  query_parameter(argc, argv, qn++,                  // Ask for apodization strength
             "\n\tApodization (Hz)? ", lwhh);
  int NO = 30;                                       // # Of Offsets (on each side)
  query_parameter(argc, argv, qn++,                  // Get # offsets
   "\n\tNumber of Positive Decoupler Offsets? ", NO);
  double offset;
  query_parameter(argc, argv, qn++,                  // Get # offsets
   "\n\tDecoupler Offset Per Step (Hz)? ", offset);
//           Set Up Variables Consistent Through All Offsets
double R = (lwhh/2)*HZ2RAD;                          // Set apodization rate
  gen_op Det = Fm(sys, IsoD);                        // Set detection operator to F-
  gen_op sigma0 = sigma_eq(sys);                     // Set density mx equilibrium
  gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);      // This is 90 detection pulse
  row_vector data(npts);                             // Block for acquisiton
//             Set Up Variables Global Over Full Profile
row_vector profile((2*NO+1)*npts, complex0);         // Block for profile
  double totaloff = double(NO)*offset;               // Total offset at end
  row_vector fidap;                                  // Block for apodized FID
  PulCycle PCyc = GP.CycGARP1(sys);                  // Empty GARP-1 pulse cycle
  SW = PCyc.FIDsync(SW);                             // Synchronize dwell times
  double td = 1/SW;                                  // Set dwell time
  double tt = (npts-1)*td;                           // Total FID length
  row_vector exp=XExponential(npts,tt,0.0,R,0);      // Block for apodization
  PCyc.print(cout, 1);
//                Loop Over Offsets, Calculate Profile
int K =0;                                            // Point index in profile
  sys.offsetShifts(-NO*offset, IsoG);                // Set 1st profile offset
  for(int ov=-NO; ov<=NO; ov++)                      // Loop over offsets
    {
    PCyc  = GP.CycGARP1(sys);                        // GARP-1 cycle this offset
    data  = PCyc.FID(npts,td,Det,sigmap);            // Acquisition this offset
    fidap = product(data,exp);                       // Apodized FID this offset
    data  = FFT(fidap);                              // Spectrum this offset
    profile.put_block(0, K, data);                   // Put spectrum in profile
    sys.offsetShifts(offset, IsoG);                  // Move system to next offset
    K += npts;                                       // Adjust profile point index
    }

  double F = totaloff + SW/2;                        // Final plot frequency
  GP_1D("prof.asc", profile, 0, -F, F);              // Output profile ASCII data
  GP_1Dplot("prof.gnu", "prof.asc");                 // Plot to screen using Gnuplot
  FM_1D("prof.mif", data,14,14,-F, F);               // Plot in FrameMaker MIF
  }
```

7.11.5 GARP-1 Modified Decoupling Profile

Suppose that we wish to follow along with the original GARP publication and reproduce all of the decoupling profiles simulated therein. How would we go about doing so? GAMMA builds up pulse cycles from a combination of a pulse waveform and a cycle overlay. For GARP-1 there is a function supplied in this module (PulGarp) which simply returns a pulse cycle based on the 25-step GARP waveform and the 4-step $R\overline{RR}R$ cycle. For decoupling sequences which are not available through an existing GAMMA function the user can simply build his/her own.

Let’s consider the “90%” GARP-1 simulation in the paper (Fig. 1c top). This was performed to examine how well GARP-1 performs when the pulses are inadvertently mis-calibrated. The authors intentionally set the decoupler field strength to be 90% of what is required for the proper pulse angles in the GARP sequence. To accomplish this in GAMMA we will modify the previous simulation by explicitly building our own (bad) GARP-1 pulse cycle. Rather that use the provided GARP-1 function, we will just scale the provided GARP waveform rf-field strength by 90% then build a modified GARP-1 pulse cycle with the scaled waveform. Everything else will be very much the same, i.e the same input values and the same input file. For convenience we will add an interactive request for the scaling factor (which will be set to 0.90 to reproduce the paper simulation).

First the simulation output.

GARP-1 90% RF-Power Decoupling Profile

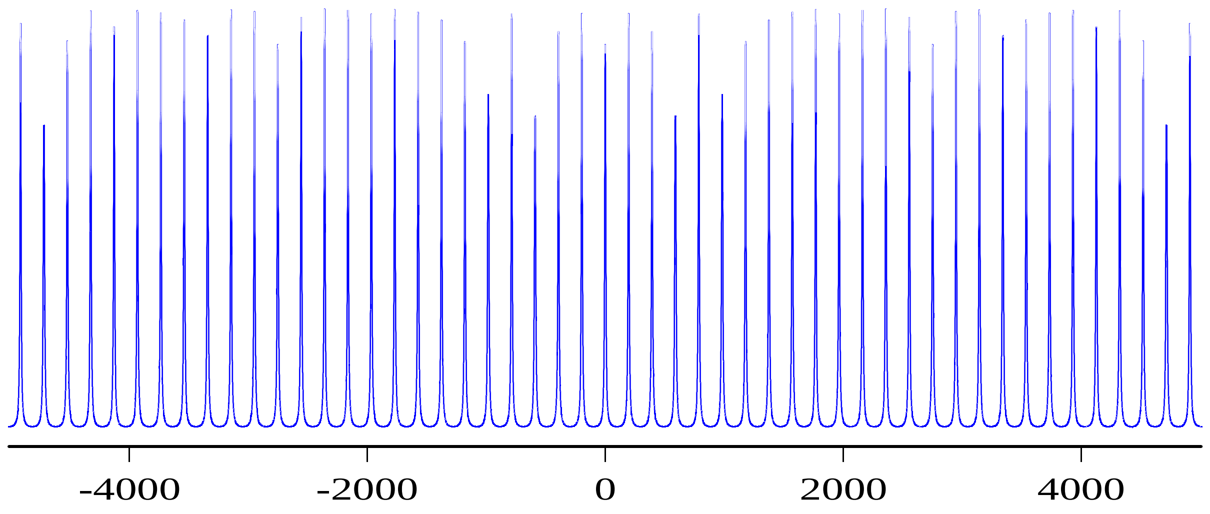


Figure 7-7 GARP-1 decoupling profile @ 90% RF power. Decoupling was performed on the carbon channel in a ^{13}C-^{1}H two spin system. The decoupling rf-field strength was set to 2 kHz x 0.90 after calibrating the pulse lengths for 2 kHz. The scalar coupling to 221 Hz. A linebroadening of 1.5 Hz was used in processing the spectra. The block size was 1K and the offset increment set to 200 Hz.

Again this is in excellent agreement with the GARP-1 paper. The simulation program is shown on the following page. I have placed the important code changes in blue so that it is evident how users can construct their own pulse cycles. I don’t know what the other decoupling sequences are in the paper cited, so I won’t bother looking at them. You can if you like though. I ran the program with the command “a.out GARPprof.sys 25 1024 1.5 25 200 0.9” where a.out is the executable.

```
/* GARPprof2.cc **************************************************************************
**                                                                                     **
**                          GAMMA Decoupling Test Program                              **
**                                                                                     **
**  This program uses the class GARP to perform a decoupling profile                  **
**  simulation.  A hard ideal pulse will be applied to a simple two                   **
**  spin  heteronuclear system.  Subsequently, an acquisition will be                 **
**  performed with GARP-1 decoupling applied on one the channel which                 **
**  is not being detected.  This process will be repeated over a range                **
**  of decoupler offsets.  The result is a GARP-1 decoupler profile.                  **
**                                                                                     **
** Author:      S.A. Smith                                                             **
** Date:        3/9/98                                                                 **
** Update:      3/9/98                                                                 **
** Version:     3.5.4                                                                  **
** Copyright:  S. Smith.  You can modify this program as you see fit for your         **
**              use, but you must leave the program intact if distrubuted.            **
**                                                                                     **
************************************************************************************** */

#include <gamma.h>

main(int argc, char* argv[])
 {
 cout << "\n\t\t\t\tGARP Decoupling Profile\n\n";
//              Read In Spin System & GARP Parameters

int qn = 1;                                     // Query index
 spin_system sys;                               // Declare a spin system
 GARP GP;                                       // GARP parameters
 String filename;                               // Input filename
 filename = sys.ask_read(argc,argv,qn++);       // Ask for/read in the system
 cout << sys;                                   // Have a look (for setting SW)
 if(sys.spins()!=2 || sys.homonuclear())
   cout << "\n\tWarning! This program has been set up for a two spin heteronuclear "
          << " system.\n Results on other systems can be unpredictable........";
 GP.read(filename);                             // Read in GARP parameters
//              Set Acquistion and Profile Parameters

 String IsoD = sys.symbol(0);                   // Detection/pulse channel
 String IsoG = GP.channel();                    // Decoupler channel
 double SW;                                     // Spectral width
 query_parameter(argc, argv, qn++,              // Get desired spectral width
              "\n\tSpectral Width (Hz)? ", SW);
 int npts = 1024;                               // Block size (must be base 2)
 query_parameter(argc, argv, qn++,              // Get block size
                   "\n\tBlock Size? ", npts);
 double lwhh = 3.0;                             // Half-height linewidth
 query_parameter(argc, argv, qn++,              // Ask for apodization strength
          "\n\tApodization (Hz)? ", lwhh);
 int NO = 30;                                   // # Of Offsets (on each side)
 query_parameter(argc, argv, qn++,              // Get # offsets
  "\n\tNumber of Positive Decoupler Offsets? ", NO);
 double offset;
 query_parameter(argc, argv, qn++,              // Get offset increment
  "\n\tDecoupler Offset Per Step (Hz)? ", offset);
 double gBsf;
 query_parameter(argc, argv, qn++,              // Get rf scaling factor
  "\n\tPulse field strength scaling factor? ", gBsf);
//              Set Up Variables Consistent Through All Offsets

 double R = (lwhh/2)*HZ2RAD;                    // Set apodization rate
 gen_op Det = Fm(sys, IsoD);                    // Set detection operator to F-
 gen_op sigma0 = sigma_eq(sys);                 // Set density mx equilibrium
 gen_op sigmap = Iypuls(sys,sigma0,IsoD, 90.);  // This is 90 detection pulse
 row_vector data(npts);                         // Block for acquisiton
//              Set Up Variables Global Over Full Profile

 row_vector profile((2*NO+1)*npts, complex0);   // Block for profile
 double totaloff = double(NO)*offset;           // Total offset at end
 row_vector fidap;                              // Block for apodized FID
 PulWaveform PWF = GP.WF();                     // GARP 25 step waveform
 PWF.scalegB1(gBsf);                            // Scale pulse waveform
 PulComposite Pcmp(PWF, sys, GP.channel());
 row_vector cyc = CYC_WALTZ4();                 // WALTZ-4 cycle overlay
 String cycname = "GARP-1 scaled";              // Modified cycle name
 PulCycle PCyc(Pcmp, cyc, cycname);             // Modified GARP-1 cycle
 SW = PCyc.FIDsync(SW);                         // Synchronize dwell times
 double td = 1/SW;                              // Set dwell time
 double tt = (npts-1)*td;                       // Total FID length
 row_vector exp=XExponential(npts,tt,0.0,R,0);  // Block for apodization
//              Loop Over Offsets, Calculate Profile

 int K =0;                                      // Point index in profile
 sys.offsetShifts(-NO*offset, IsoG);            // Set 1st profile offset
 for(int ov=-NO; ov<=NO; ov++)                  // Loop over offsets
  {
  Pcmp = PulComposite(PWF, sys, GP.channel());// Reset GARP composite pulse
  PCyc = PulCycle(Pcmp, cyc, cycname);          // Reset modified GARP-1 cycle
  data  = PCyc.FID(npts,td,Det,sigmap);         // Acquisition this offset
  fidap = product(data,exp);                    // Apodized FID this offset
  data  = FFT(fidap);                           // Spectrum this offset
  profile.put_block(0, K, data);                // Put spectrum in profile
  sys.offsetShifts(offset, IsoG);               // Move system to next offset
  K += npts;                                    // Adjust profile point index
  }
 double F = totaloff + SW/2;                    // Final plot frequency
 GP_1D("prof.asc", profile, 0, -F, F);          // Output profile ASCII data
 GP_1Dplot("prof.gnu", "prof.asc");             // Plot to screen using Gnuplot
 FM_1D("prof.mif", profile ,14,14,-F, F);       // Plot in FrameMaker MIF
 }
```

7.11.6 GARP Decoupling With Relaxation

Now lets do something a bit more exotic with GAMMA and GARP. Here we will add in the effects of relaxation while the decoupler is on. Since we already have a program that simulates GARP-1 decoupled spectra (GARPdec1.cc) without relaxation, we need only make the proper modifications to that program and its input in order to obtain the simulation we want.

Lets review a few of basic changes we'll need. First, rather than working with an isotropic spin system (spin_system) in our program, we need to work with a oriented spin system that is moving isotropically. That is, a spin system that keeps track of dipolar, CSA, and quadrupolar tensors for all spins or spin-pairs. Thus we need to replace ***spin_system*** with ***sys_dynamic***. Second, when the system is read in from an external ASCII file it will look for tensor quantities as well as dynamical values (correlation times). Next we will have to create a relaxation matrix and Liouvillian that defines how the system evolves. And lastly, we'll have to use an FID function that includes that evolves under the defined Liouvillian so that relaxation (and exchange) are accounted for.

Now that all might sound rather complicated, but it actually requires only minor adjustments to the program we already have at our disposal. Have a look. The code on the left was clipped out of the program GARPdec0.cc covered earlier in this chapter and the code on the right the modifications we need to include relaxation effects. I've left out some of the comments.

```
.                                                        .
.                                                        .
.                                                        .
spin_system sys;          // Isotropic spin system       sys_dynamic sys;         // New system type
.                                                        .
.                                                        .
.                                                        .
gen_op H = Ho(sys);       // Isotropic Hamiltonian       gen_op H = Ho(sys);      // Isotropic Hamiltonian
.                                                        super_op L = RDD(sys,H); // Dipolar relaxation.
.                                                        .
.                                                        .
PulCycle PCyc = GP.CycGARP1(sys);                        PulCycle PCyc = GP.CycGARP1(sys, L);
.                                                        .
.                                                        .
.                                                        .
row_vector data = PCyc.FID(npts,td,Det,sigmap);          row_vector data = PCyc.FIDR(npts,td,Det,sigmap);
.                                                        .
.                                                        .
.                                                        .
```

Now that wasn't so bad was it? We generate a dipole-dipole relaxation superoperator, called "L" in the above code, and include it in the function calls which generate the GARP pulse cycle. Then we call an FID function which will include relaxation effects. Keep in mind that L resides in spin Liouville space and is typically big. Running decoupling on 5 spins will take a while and it gets worse the larger the spin system. It is a "Redfield" relaxation superoperator dealing with coupled relaxation effects (and not just longitudinal relaxation either...).

Now on to our program. The following page contains the modifications shown. I do build "L" in a different fashion than indicated above so that I can include multiple relaxation effects and exchange if I desire, rather than just setting it to only dipolar relaxation. This will become a little more clear when you look at the simulation output *versus* the input ASCII parameter file........

7.12 Chapter Source Codes

GarpWF0.cc

```
/* GarpWF0.cc ********************************************************************   **
**                                                                                   **
**                     GAMMA GARP Simulation Example Program                         **
**                                                                                   **
** This program examines the basic GARP sequence Waveform.  I does no                **
** NMR computations involving GARP, it merely spits out plots so that                **
** the default GARP (GARP-1) sequence can be readily viewed.                         **
**                                                                                   **
** Assuming a.out is the executable of this program, then the following              **
** command will generate a single 25 step GARP-1 waveform which will                 **
** be displayed on screen if Gnuplot is available.  It will also make                **
** an editable FrameMaker MIF file of the waveform.                                  **
**                                                                                   **
**                         a.out                                                     **
**                                                                                   **
** Author:    S.A. Smith                                                             **
** Date:      2/27/98                                                                **
** Copyright: S.A. Smith, February 1998                                              **
**                                                                                   **
***************************************************************************************/

#include <gamma.h>                                    // Include GAMMA

main(int argc, char* argv[])
  {
  cout << "\n\n\t\t\tGAMMA GARP Waveform Program 0\n";
  GARP GP(500.0, "1H");                               // Set GARP parameters
  PulWaveform PWF = GP.WF_GARP1();                    // Construct waveform
  PWF.GP(1, 1, 5);                                    // Plot waveform(s), gnuplot
  PWF.FM(1, 1, 5);                                    // Plot waveform(s), Framemaker
  cout << "\n\n";                                     // Keep screen nice
  }
```