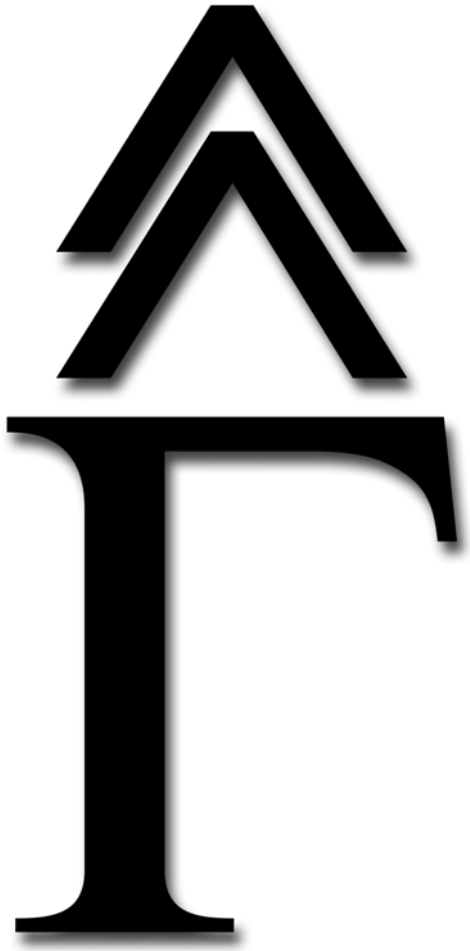
GAMMA

Level 2 Module



Level 2 Module

Table of Contents

1 Introduction

This module, Level 2, contains classes and functions which facilitate magnetic resonance simulations. These don't fall into any particular categories, they are grouped together only because they fit into GAMMA at this relative level of complexity. Much of the functionality supplied in this module is used in subsequent (higher level) GAMMA classes and modules.

Contents of the module are as follows.

- ***TrnsTable1D*** -This class maintains a table of transitions. Each table is able to generate either time-domain or frequency domain spectra (NMR and EPR) and allows the user to manipulate the table prior to spectrum generation.
- ***Acquire1D*** - This class allows for rapid generation of repetetive expectation values in density operator simulations. It has the ability to generate transition tables and interfaces with the class in TrnsTable1D. Density operator evolution must occur under a constant Hamiltonian (or Liouvillian).
- ***EAngles*** - This class defines a set of Euler angles, $\{\alpha, \beta, \gamma\}$. These angles are used to define rotations in other areas of the GAMMA platform. The angles restricted to the ranges α, $\gamma = [0, 360)$ and $\beta = [0, 180]$.
- ***Quatern*** - This class embodies a Quaternion, a set of 4 values ***{A, B, C, D}*** that defines a rotation. Quaternions are very useful in generating composite rotations. They interact with GAMMA Euler Angles (class EAngles) so that composite rotations may be defined by a single set of such angles.
- ***BaseDecomp*** - This class allows users to decompose a density operator in terms of a defined basis. The user may specify the base in which the decomposition is to be performed or one of the provided decompositions may be used.

2 1D Transitions Table

2.1 Overview

The class ***TTable1D*** facilitates the storage and manipulation of simple 1D spectra in GAMMA simulations. Many 1D spectra may be distilled down to a list of transitions along with there associated frequencies, linewidths, intensities, and phases. Indeed, GAMMA acquisition classes (e.g. see class ***acquire1D***) which uses density operator formalism to generate spectra produces a transitions table.

2.2 Available Functions

2.3 Document Sections

2.4 Constructors and Assignment

2.4.1 TTable1D

Usage:

```
#include <TrnsTable1D.h>
TTable1D()
TTable1D(const matrix& mx);
TTable1D(const TTable1D& TT1)
```

Description:

The function *TTable1D* is used to create a 1D transitions table.

1. TTable1D() - Creates an "empty" NULL transitions table. Can be later filled by an assignment.
2. TTable1D(const matrix& mx) - Sets up transitions table from the input array ***mx***. Note that transition tables have only 2 columns and will restrict the input array to 2 columns as well.f
3. TTable1D(const TTable1D&TT1) - Constructs a new table identical to the input TT1.

Note that construction of transition tables sets up may default parameters which influence table output and performance in generation of spectra. A simple example would be whether the table outputs frequencies in Hz vs. PPM. Users should note the access functions to see which defaults need adjustment.

Return Value: TTable1D

Examples:

```
include <gamma.h>
int main ()
  {
  TTable1D TT;                    // A null transitions table
  matrix mx(2,2, complex0);       // An array we'll use to make a table
  TTable1D TT1(mx);               // Construct table
  TTable1D TT3(TT1);              // Another copy of TT1
  }
```

See Also: =

2.4.2 =

Usage:

```
#include <TrnsTable1D.h>
void TTable1D operator = (const TTable1D& TT1)
```

Description:

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

Return Value: None, the function is void

Example:

```
include <gamma.h>
```

```
int main ()
  {
  matrix mx(10,2);
  TTable1D TT1(mx);
  TTable1D TT3 = TT1;
  }
```

See Also: TTable1D

2.5 Access Functions

2.5.1 center

Usage:

```
#include <Level2/TrnsTable1D.h>
double TTable1D::center()
```

Description:

Function ***center*** returns the center of transitions defined by $(\omega_{max} - \omega_{min})/2$. The value in is radians/sec.

Return Value: double

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                      // An empty transitions table
  .......                           // Fill the table somehow
  double Wcenter = TT.center();     // Get the center of all transitions
  }
```

See Also: offset

2.5.2 offset

Usage:

```
#include <Level2/TrnsTable1D.h>
void TTable1D::offset(double F, int inHz=1)
void TTable1D::offset(double F, int tr, int inHz)
```

Description:

The function ***offset*** allows the user to offset all or individual transition frequencies in the table. The frequency offset is ***F*** and the flag ***inHz*** indicates whether ***F*** has been input in Hertz (***inHz!=0***) or in rad/sec (***inHz=0***). If a transition index, ***tr***, is given then only that transition is offset.

Return Value: Void

Example:

```
include <gamma.h>
int main ()
  {
  TTable1D TT;                      // An empty transitions table
  .......                           // Fill the table somehow
  TT.offset(100.0);                 // Offset all transitions by 100 Hz
  TT.offset(100.0, 1, 0),           // Offset 2nd transition by 100 radians/sec
  }
```

See Also: center

2.5.3 FRscale

Usage:

```
#include <Level2/TrnsTable1D.h>
void TTable1D::FRscale(double Fscf)
void TTable1D::FRscale(double Fscf, int tr)
```

Description:

The function ***FRscale*** allows the user to scale all or individual transition frequencies in the table. The frequency scaling factor is ***Fscf***. If a transition index, ***tr***, is given then only that transition is scaled. Scaling implies multiplication of the existing frequency value(s) by ***Fscf***.

Return Value: void

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // An empty transitions table
  .......                         // Fill the table somehow
  TT.FRscale(2);                  // Double all frequencies
  TT.FRscale(-3,0);               // Multiply 1st transition frequency by -3
  }
```

See Also: IScale

2.5.4 IScale

Usage:

```
#include <Level2/TrnsTable1D.h>
void TTable1D::Iscale(double Iscf)
void TTable1D::Iscale(double Iscf, int tr)
```

Description:

The function ***Iscale*** allows the user to scale all or individual transition intensities in the table. The frequency scaling factor is ***Iscf***. If a transition index, ***tr***, is given then only that transition is scaled. Scaling implies multiplication of the existing intensity value(s) by ***Fscf***.

Return Value: void

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // Declare TTable1D Parameters
  .......                         // Fill the table somehow
  TT.Iscale(2);                   // Double all intensities
  TT.Iscale(-3,0);                // Multiply 1st transition intensity by -3
  }
```

See Also: FRScale

2.5.5 broaden

Usage:

#include <Level2/TrnsTable1D.h>

void TTable1D::broaden(double LWR, int inHz=1)

void TTable1D::broaden(double LWR, int tr, int inHz=1)

Description:

The function ***broaden*** allows the user to linebroaden/apodize all or individual transition intensities in the table. The broadening value is ***LWR.*** If a transition index, ***tr***, is given then only that transition is broadened. The value of ***LWR*** is added to any existing linewidth values. The flag ***inHz*** indicates what the units of ***LWR*** are and how it is to be added. If input in Hertz (***inHz!=0***) LWR Hz linewidth is applied. If input in rad/sec (***inHz=0***) the ***LWR*** is added to the existing transition decay rate.

Return Value: void

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // Declare TTable1D Parameters
  .......                         // Fill the table somehow
  TT.broaden(2);                  // Add 2 Hz linebroadening to all transitions
  TT.Iscale(0.1,0);               // Add 0.2/sec to decay rate R of 1st transition
  }
```

2.5.6 resolution

Usage:

#include <Level2/TrnsTable1D.h>

void TTable1D::resolution(double res)

Description:

The function ***resolution*** adjusts the table to reflect a specified resolution ***res***. Any transitions in the table will be blended if their frequencies are with ***res*** rad/sec from each other. This will only occur if their linewidths and phase match. This will affect the table itself, not just the printed output of the transactions.

Return Value: Void

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // Declare TTable1D Parameters
  .......                         // Fill the table somehow
  TT.resolution(.002);            // Combine transitions within 0.002 rad/sec
  }
```

See Also:

2.6 Spectrum Generation Functions

2.6.1 T

2.6.2 Ts

Usage:

```
#include <Level2/TrnsTable1D.h>
row_vector TTable1D::T(int npts, double tinc) const
void TTable1D::T(row_vector data, double tinc) const
vector<row_vector> TTable1D::Ts(int npts, double tinc)
```

Description:

Function ***T*** generates a time domain spectrum by summing over decaying exponentials associated with each transition. The points will be separated in time by ***tinc*** seconds. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned.

Return Value: void or row_vector

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // Declare TTable1D Parameters
  TT.read("filein.pset")          // Read in TTable1D Parameters
  TTable1D GWF = TT.WF();         // Make TTable1D waveform (TTable1D-1)
  }
```

See Also: F, Fs, FD

2.6.3 TD

Usage:

```
#include <Level2/TrnsTable1D.h>
row_vector TTable1D::TD(int npts, double tinc) const
void TTable1D::TD(row_vector data, double tinc) const
```

Description:

Function ***TD*** generates a time domain spectrum by summing over decaying differential exponentials associated with each transition. The points will be separated in time by ***tinc*** seconds. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned.

Return Value: void or row_vector

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                    // Declare TTable1D Parameters
  TT.read("filein.pset")          // Read in TTable1D Parameters
  TTable1D GWF = TT.WF();         // Make TTable1D waveform (TTable1D-1)
  }
```

See Also: F, Fs, FD

2.6.4 F

2.6.5 Fs

Usage:

#include <Level2/TrnsTable1D.h>

row_vector TTable1D::F(int npts, double Fst, double Ffi) const

void TTable1D::F(row_vector data, double Fst, double Ffi) const

Description:

Function ***F*** generates a frequency domain spectrum by summing over complex Lorentzians associated with each transition. The output will span the frequency range [***Fst, Ffi***] where the two frequencies are input in Hz. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned.

Return Value: void or row_vector

Example:

```
#include <gamma.h>
int main ()
  {
  spin_system sys;                    // Declare a spin system
  sys.read("filein.sys");             // Read in the spin system
  TTable1D TT;                        // Declare TTable1D Parameters
  TT.read("filein.pset")              // Read in TTable1D Parameters
  PulComposite GCP = TT.PCmp(sys);    // TTable1D composite pulse(TTable1D-1)
  }
```

See Also: WF, CycTTable1D1

2.6.6 FD

Usage:

#include <Level2/TrnsTable1D.h>

row_vector TTable1D::FD(int npts, double Fst, double Ffi) const

void TTable1D::FD(row_vector data, double Fst, double Ffi) const

Description:

Function ***FD*** generates a frequency/field domain spectrum by summing over complex differential Lorentzians associated with each transition. The output will span the range [***Fst, Ffi***] where the two frequencies are input in Hz. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned.

Return Value: void or row_vector

Example:

```
#include <gamma.h>
int main ()
  {
  spin_system sys;                    // Declare a spin system
  sys.read("filein.sys");             // Read in the spin system
```

```
TTable1D TT;                                  // Declare TTable1D Parameters
TT.read("filein.pset")                        // Read in TTable1D Parameters
PulCycle GCy = TT.CycTTable1D1(sys); // TTable1D-1 pulse cycle
```

See Also: WF, PCmp

2.7 Auxiliary Functions

2.7.1 R2

2.7.2 Fr

2.7.3 I

Usage:

```
#include <Level2/TrnsTable1D.h>
double TTable1D::R2(int tr) const
double TTable1D::Fr(int tr) const
complex TTable1D::I(int tr) const
```

Description:

The function ***R2*** will return the decay rate, R_2, of transition ***tr*** in rad/sec. The function ***Fr*** will return the frequency, ω, of transition ***tr*** in rad/sec.The function ***I*** returns the intensity (complex) of transition ***tr.***

Return Value: double or complex

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                          // Declare TTable1D Parameters
  .......                               // Fill the table somehow
  double R2 = TT.Fr(0);                 // Rate of 1st transition
  double W = TT.Fr(6);                  // Frequency of 7th transition
  double I = TT.Fr(3);                  // Intensity of 4th transition
  double N = norm(I);                   // Magnitude of 4th transition
  double Phi = phase(I);                // Phase of 4th transition
  }
```

2.7.4 size

Usage:

```
#include <Level2/TrnsTable1D.h>
int TTable1D::size()
```

Description:

The function ***size*** returns the number of transitions in the table.

Return Value: integer

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                          // Declare TTable1D Parameters
  .......                               // Fill the table somehow
  cout << "\n\t" << TT.size() << " Transitions\n";
```

```
}
```

See Also:

2.7.5 Frmax

2.7.6 Frmin

2.7.7 Tdmin

2.7.8 LWmax

2.7.9 Imax

Usage:

```
#include <Level2/TrnsTable1D.h>
double TTable1D::Frmax()
double TTable1D::Frmin()
double TTable1D::Tdmin()
double TTable1D::LWmax()
double TTable1D::Imax()
```

Description:

These functions return maxima and minima values for the table. ***Frmax*** and ***Frmin*** return the maximum and minimum frequencies in the table respectively. Tdmin returns the minimum dwell time required in order to satisfy a Nyquist sampling rate. LWmax returns the biggest linewidth present. Imax returns the largest intensity (norm) of all transitions.

Return Value: double

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;                          // Declare TTable1D Parameters
  }
```

See Also:

2.7.10 sum

Usage:

```
#include <Level2/TrnsTable1D.h>
complex TTable1D::value(int i)
```

Description:

The function ***value*** returns a complex number for the values which define the pulse waveform step ***i***. The value contains the number $\{\gamma B1, \phi\}$, 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 complex number.

Example:

#include <gamma.h>

See Also:

2.7.11 Sort

Usage:

#include <Level2/TrnsTable1D.h>
double TTable1D::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: double.

2.8 Formatted Output

Transition table can be output in two different forms, both are ASCII. The first is intended to be viewable on screen, the second is intended to be used as a GAMMA parameter file. While the latter has few output options, the former has several flags which will affect output appearance

2.8.1 setType

Usage:

#include <Level2/TrnsTable1D.h>
void TTable1D::setType(int typ)

Description:

The function ***setType*** is used to set an internal flag which indicates the type of frequency data in the table. The values of ***typ*** are ***0=Hz***, ***1=PPM***, ***2=Gauss***.

Return Value: void

The function returns a double.

Example:

```
#include <gamma.h>
int main ()
  {
  }
```

See Also:

2.8.2 setSort

Usage:

#include <Level2/TrnsTable1D.h>
void TTable1D::setSort(int i)

Description:

The function ***setSort*** sets an internal flag that triggers whether to sort the transitions by frequency when the table is output. If ***i=0*** then the transitions remain in their current order whereas if *i!=0* the transitions will be output from high frequency to low frequency. Note that by default, sorting will be ON - so typically this function isn't used.

Return Value:

The function is void.

Example:

```
#include <gamma.h>
int main ()
  {
  .....                          // Generate TTable TT
  TT.sort(1);                    // Set for frequency sorting
  cout << TT;                    // Output table, transitions sorted
```

```
TT.sort(0);                    // Set for no frequency sorting
cout << TT;                    // Output table, transitions unsorted
}
```

2.8.3 setConv

Usage:

#include <Level2/TrnsTable1D.h>
void TTable1D::setConv(double cf)

Description:

The function ***setConv*** sets an internally stored conversion factor to the input value ***cf***. The conversion factor is used during table output to switch frequency units to those desired by the user. There are currently on two instances frequency output types which use this conversion factor. 1.) When the user desires frequencies to be output in PPM then the conversion factor must be Hz->PPM, i.e. the Larmor frequency in MHz. That is, for a 400 MHz spectrometer, the conversion factor should be 400 if the transitions in the table are proton transitions. 2.) When the table contains ESR transitions the frequencies need to be converted to Gauss. This is done by the g-factor.

Return Value: void

Example:

```
#include <gamma.h>
int main ()
  {
  .                                  // Generate TTable TT for protons
  .                                  // from sim. involving system sys
  TT.setType(2);                     // Set frequency output in PPM
  TT.setConv(sys.Omega());           // Set conversion as proton Larmor
  cout << TT;                        // Output table, frequencies in PPM
  }
```

2.8.4 setIcut

Usage:

#include <Level2/TrnsTable1D.h>
void TTable1D::setIcut(double cutoff)

Description:

The function ***setIcut*** sets an internally stored intensity ***cutoff*** value. This value is used both in writing the table to an output stream as well as in the generation of spectra. Any transition having an intensity below the value set will not be included in spectral output. The cutoff value has a default setting of 1.e-12 upon construction. Using this function one is able to ignore weak/forbidden transitions in transition table output and/or generation of 1D spectra. Alternatively, the function ***status*** allows users to see the current cutoff setting.

Return Value:

The function is void.

Example:

```
#include <gamma.h>
int main ()
  {
  .                                  // Generate TTable TT for protons
  .                                  // from sim. involving system sys
  TT.setType(2);                     // Set frequency output in PPM
  TT.setConv(sys.Omega());           // Set conversion as proton Larmor
```

```
cout << TT;                          // Output table, frequencies in PPM
}
```

2.8.5 setSN

Usage:

```
#include <Level2/TrnsTable1D.h>
void TTable1D::setSN(double S2N)
```

Description:

The function ***setSN*** sets an internally stored intensity ***signal to noise*** value. This value, which defaults to 0 for no noise, is used to add random noise to spectra generated by the table.

Return Value:

The function is void.

Example:

```
#include <gamma.h>
int main ()
  {
  .                                  // Generate TTable TT for protons
  .                                  // from sim. involving system sys
  TT.setType(2);                     // Set frequency output in PPM
  TT.setConv(sys.Omega());           // Set conversion as proton Larmor
  cout << TT;                        // Output table, frequencies in PPM
  }
```

2.8.6 setRprint

2.8.7 setLWprint

2.8.8 setT2print

2.8.9 setPHprint

Usage:

```
#include <Level2/TrnsTable1D.h>
void TTable1D::setRprint(int pr);
void TTable1D::setLWprint(int pr);
void TTable1D::setT2print(int pr);
void TTable1D::setPHprint(int pr);
```

Description:

These functions can be used to control which columns are printed during formatted output. The relationship between the function and column are obvious: ***setRprint*** controls whether transition rates are printed, ***setLWprint*** controls whether linewidths column is printed, ***setT2print*** controls whether T_2 values are printed, and ***setPHprint*** controls whether the phases are printed. The input flag ***pr*** should be set as follows: ***>0 == print***, ***0 == print as needed***, ***<0 == do not print.***

Return Value: void

Example:

```
#include <gamma.h>
```

```
int main ()
  {
  .                                  // Generate TTable TT for protons
  .                                  // from sim. involving system sys
  TT.setType(2);                     // Set frequency output in PPM
  TT.setConv(sys.Omega());           // Set conversion as proton Larmor
  cout << TT;                        // Output table, frequencies in PPM
  }
```

2.8.10 print

2.8.11 <<

Usage:

#include <Level2/TrnsTable1D.h>

ostream& TTable1D::print(ostream& ostr)

ostream& operator << (ostream& ostr, TTable1D& TT)

Description:

The function print and the operator **<<** add the TTable1D parameters specified as an argument ***TT*** to the output stream ***ostr***.

Return Value: Output Stream

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;
  TT.read("filein.pset");
  TT.print(cout);
  cout << TT;
  }
```

See Also: none

2.9 Formatted Input

Transition tables can be read in from an external ASCII file which is in GAMMA parameter set format. Parameters recognized by the transition table are discussed in Section 2.13.

2.9.1 readPSet

Usage:

```
#include <Level2/TrnsTable1D.h>
bool TTable1D::readPSet(const string& filein, int indx=-1, int warn=2)
bool TTable1D::readPSet(const ParameterSet& pset, int indx, int warn=2)
```

Description:

The function ***readPSet*** will attempt to fill the TTable1D values from parameters found in either the external ASCII file ***filein*** or from those found in the ParameterSet ***pset***. When read in from a file, the file must be in GAMMA parameter set format. An optional prefix number, ***indx***, can be specified to force use on parameter names preceded with ***[indx]***. The function returns true/false depending on whether enough parameters have been found to fill up a valid transitions table. A warning level, ***warn***, can be specified to indicate what to do upon failure: 0=take no action, 1=issue warnings, 2=issue fatal errors and stop program execution.

Return Value: True/False

Example:

```
#include <gamma.h>
int main ()
  {
  TTable1D TT;
  TT.read("filein.pset");
  TT.print(cout);
  }
```

See Also: write

2.10 Auxiliary Functions

2.10.1 FZ

Usage:

#include <TrnsTable1D.h>
int TTable1D::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 <gamma.h>
```

See Also:

2.11 Examples

2.11.1 Simple NMR

This is a simple program that demonstrates the basics of using a transitions table to generate an NMR spectrum. In this example the transitions table will be built by hand and three “equivalent’ spectra output: 1.) An FID 2.) A Spectrum and 3.) A Table of Transitions. Later examples will show how more complicated transition tables can be read in from external ASCII files, but keep in mind that normally some simulation will spit out the transition table...

2.12 Description

2.12.1 Introduction

Class ***TTable1D*** is designed to facilitate the storage and manipulation of 1-dimensional spectra in GAMMA MR simulation programs. Note that within the context of nuclear magnetic resonance simulations, class ***acquire1D*** is used because it has the functionality of ***TTable1D*** and ties into the density operator formalism used in such calculations. A single transition is taken to consist of a frequency, decay rate, and a complex intensity (magnitude and phase).

$$tr = \{ \omega, R, I \}$$

A transition table is then just a list of many transitions, conveniently packed into an Nx2 complex matrix where N is the number of transitions in the transitions table.

Transition Table Matrix Elements

Index	***R***	ω	***I***	
0	***Re<0|mx|0>***	***Im<0|mx|0>***	***Re<0|mx|1>***	***Im<0|mx|1>***
1	***Re<1|mx|0>***	***Im<1|mx|0>***	***Re<1|mx|1>***	***Im<1|mx|1>***
2	***Re<2|mx|0>***	***Im<2|mx|0>***	***Re<2|mx|1>***	***Im<2|mx|1>***
. . .	. . .	. . .	. . .	. . .
N-1	***Re<N-1|mx|0>***	***Im<N-1|mx|0>***	***Re<N-1|mx|1>***	***Im<N-1|mx|1>***

2.12.2 NMR Spectra

Each transition in the table can be used to generate either a decaying complex exponential function or a complex Lorentzian function. The decaying exponential (see Level 1 module exponential documentation) is given by

$$Exp(tr, t) = I \times e^{i(\omega_{tr} + iR)t} = I[\cos(\omega_{tr}t) + i\sin(\omega_{tr}t)]e^{-Rt} \quad (2\text{-}1)$$

and the Lorentzian as (see Level 1 module Lorentzian documentation)

$$L(tr, \omega) = I \times \frac{R - i(\omega - \omega_{tr})}{R^2 + (\omega - \omega_{tr})^2} = \frac{I}{R + i(\omega - \omega_{tr})} \quad (2\text{-}2)$$

The two functions are Fourier transform pairs. By summing over transitions, one can generate ei-

ther a time domain spectrum (akin to an FID in NMR) as

$$FID(t) = \sum_{k=0}^{N-1} I_k \times e^{i(\omega_k + iR_k)t} = \sum_{k=0}^{N-1} I_k e^{-R_k t} e^{i\omega_k t} \tag{2-3}$$

or a frequency domain spectrum (the Fourier transform of the time domain spectrum) from

$$S(\omega) = \sum_{k=0}^{N-1} I_k \times \frac{R_k - i(\omega - \omega_k)}{R_k^2 + (\omega - \omega_k)^2} = \sum_{k=0}^{N-1} I_k \times \frac{1}{R_k + i(\omega - \omega_k)} \tag{2-4}$$

When related to NMR, the ***R*** values are decay rates (1/sec) and are directly related to the Lorentzian half-height linewidth ***lwhh*** and NMR T_2 relaxation time as

$$R = \frac{1}{T_2} = 0.5 lwhh \tag{2-5}$$

2.12.3 EPR Spectra

A analogous treatment can be used in generating EPR spectra with a transition table. In this instance, due to CW detection methods, it is differential exponentials and differential Lorentzians that associated with each transition. The time domain differential exponential is given by (see Level 1 module exponential documentation)

$$DExp(tr, t) = I \times -ite^{i(\omega_{tr} + iR)t} \tag{2-6}$$

and its Fourier transform, a differential Lorentzian, is given by(see Level 1 module Lorentzian documentation)

$$DL(tr, \omega) = \frac{-iI}{[R + i(\omega - \omega_{tr})]^2} \tag{2-7}$$

By summing over transitions, one can generate either a time domain spectrum (akin to an FID in NMR but not often used in EPR due to CW measurement) as

$$DFID(t) = \sum_{i=0}^{N} -iI_i \times te^{i(\omega_i + iR)t} = \sum_{i=0}^{N} -iI_i e^{-R_i t} e^{i\omega_i t} \tag{2-8}$$

or a frequency (or field) domain spectrum (the Fourier transform of the time domain spectrum)

from

$$DS(\omega) = \sum_{k=0}^{N-1} \frac{-iI_k}{[R_k + i(\omega - \omega_k)]^2} \tag{2-9}$$

When related to EPR, the ***R*** values are decay rates (1/sec) and are directly related to the differential Lorentzian peak-to-peak linewidth ***lwpp*** and EPR T_2 relaxation time as

$$R = \frac{1}{T_2} = \sqrt{\frac{3}{4}} lwpp \tag{2-10}$$

Since EPR transitions are often expressed in field (or g-values) rather than in frequency units, the user must occasionally apply some conversion factors when using transition tables to track EPR spectra. See the examples on how to do this.

2.13 TTable1D Parameters

This section describes how an ASCII file may be constructed that is self readable by a TTable1D variable. The file can be created with an editor of the users choosing and is read with the TTable1D member function “read”. This provides for an extremely flexible and program independent means of implementing TTable1D in NMR/EPR simulations. The TTable1D (ASCII) input file is scanned for the specific parameters which specify the transition values[1]: $\{\omega, R, I\}$, and the total number of transitions. These parameters are recognized by certain keywords, as shown in the following table.

Transitions Table Parameters

Parameter Keyword	Assumed Units	Examples Parameter (Type) : Value - Statement			
TrN	none	TrN	(0) :	16	- Number of Transitions
TrF	Hz	TrF(0)	(1) :	600.0	- 1st Transition Frequency (Hz)
TrPPM	PPM	TrPPM(6)	(1) :	7.0	- 7th Transition Frequency (PPM)
TrW	1/sec	TrW(6)	(1) :	1234.75	- 7th Transition Frequency (rad/sec)
TrB	Gauss	TrB(2)	(1) :	12.756	- 3rd Transition Field (Gauss)
TrR	1/sec	TrR(0)	(1) :	0.3	- 1st Transition Rate (rad/sec)
TrLW	Hz	TrLW(1)	(1) :	0.2	- 2nd Transition Linewidth[a] (Hz)
TrPP	Gauss	TrPP(1)	(1) :	0.2	- 2nd Transition Linewidth[b] (Gauss)
TrT2	sec	TrT2(2)	(1) :	1.1	- 3rd Transition T2 (sec)
TrSN	none	TrSN	(1) :	200.5	- Signal To Noise Ratio
TrI	none	TrI(0)	(1) :	2.0	- 1st Transition Intensity
TrPh	degrees	TrPh(0)	(1) :	0.0	- 1st Transition Phase (degrees)

a. This is the linewidth at half-height, the value commonly used to describe Lorentzians.
b. This is the peak-to-peak linewidth, the value commonly used to describe differential Lorentzians.

The order in which these parameters reside in the ASCII file is of no consequence. Transition phases are always optional and set to zero if not specified. Transition intensities MUST be specified and these are the magnitudes of the internally stored values. The transition linewidth is optional and set to zero if unspecified. The set { TrR, TrLW, TrPP, TrT2) are redundant. If multiple definitions are found the precedent is TrLW > TRPP> TrT2 > TrR. The set { TrF, TrPPM, TrW, TrB } are also redundant. Their priorities are TrPPM > TrF > TrW > TrB.

1. The ASCII file must contain viable parameters in GAMMA format. It is a GAMMA parameter set and, as such, may contain any amount of additional information along with the valid TTable1D parameters.

2.13.1Internal Structure

Class ***TTable1D*** is derived from class ***matrix***, however this is done so that matrix is private and not directly accessible. This is because the matrix must be complex and of dimension Nx2 where N is the number of transitions in the array. Other than the matrix, the class consists only of flags and conversion factors which affect output.

Transitions Table Contents

Name	Type	Utility
this	matrix	Stores transitions:
ICUT	double	Intensity cutoff (norm
INORM	double	Intensity Normalization Factor
PERCUT	double	Signal Intensity Percentage Cutoff
SN	double	Signal to Noise Ration
FRQTYPE	int	Frequency Type Flag
FRQSORT	int	Frequency Sort Flag
FRQCONV	double	Frequency Conversion Factor
HP	int	Header Print flag (affects output only)
RP	int	R Rate Print flag (affects output only)
LWP	int	Linewidth Print flag (affects output only)
T2P	int	T2 Print flag (affects output only)
PHP	int	Phase Print flag (affects output only)

However, mostly because class ***acquire1D*** supports both Hilbert space and Liouville space computations, the structure contains additional entities beyond those explicit in this equation. These are given in the following table.

3 Class Acquire1D

3.1 Overview

The class ***Acquire1D*** contains the computational core necessary for repeatedly determining expectation values according to

$$\langle Op(t) \rangle = Trace\{ Op * \sigma(t) \} = \langle Op^{\dagger} | \sigma(t) \rangle$$

when the system evolves either in Hilbert space as $\sigma(t) = e^{-iHt}\sigma_o e^{iHt} = U\sigma_o U^{-1}$ or in Liouville space in accordance with $|\sigma(t)\rangle = \Gamma(t)|\sigma_o\rangle$

in Liouville space. Here ***Op*** is an operator (for a physical quantity), ***L*** the system Liouvillian, and ***Op(t)*** is another operator evolving in time under a static Hamiltonian. Use of Acquire can increase computational speed and reduce code complexity. The class is small with few functions; it serves only to facilitate a commonly performed calculation.

3.2 Available Functions

3.3 Covered Acquisition Theory

3.4 Constructors & Assignment

3.4.1 acquire1D

Usage:

```
#include <Level2/acquire1D.h>
void acquire1D()
void acquire1D(gen_op &D, gen_op& H, double cutoff=1.e-12)
void acquire1D(gen_op &D, super_op& L, double cutoff=1.e-12)
void acquire1D(gen_op &D, super_op& L, gen_op& siginf, double cutoff=1.e-12)
void acquire1D(const acquire1D &ACQ)
```

Description:

The function ***acquire1D*** is used to create a acquisition computational core. The general operator ***D*** is set as the detection operator and the optional value ***cutoff*** may be specified as where to assume zero transition intensity. If a general operator ***H*** is input following ***D*** it is assumed to be a static evolution Hamiltonian and will trigger a computation in Hilbert space. If instead a superoperator L is input following D it is assumed to be a system Liouvillian and will trigger a computation in Liouville space. Finally, if general operator ***siginf*** follows any input superoperator it will be taken as the infinite time density operator, i.e. that which the system evolves to at infinite time. Without any argumets an empty ***acquire1D*** is produced. With an acqure1D as an argument, ***ACQ***, a copy of the input is created.

Return Value: none

Examples:

```
#include <gamma.h>
main()
  {
  .....                          // Generate operators D and H
  .....                          // Generate superoperator L
  .....                          // Generate infinite time density operator siginf
  acquire1D ACQ;                 // An empty acqure1D
  acquire1D AHS(D,H);            // Acquisiton evolving under H, detection D
  acquire1D ALS(D,L);            // Acquisiton evolving under L, detection D
  acquire1D ABWR(D,L,siginf);    // Acquisiton evolving under L towards siginf, detection D
  acquire1D A(ALS);              // Acquisiton A is a copy of ALS
  }
```

See Also: =

3.4.2 =

Usage:

```
#include <Level2/acquire1D.h>
void acquire operator = (acquire &ACQ)
```

Description:

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

Return Value: Void

Example:

```
#include <gamma.h>
main()
  {
  .....                          // Generate an Acquire1D variable ACQ
  acquire1D A2 = ACQ;            // Set A2 equal to ACQ
  }
```

See Also: acquire1D

3.5 Access Functions

3.5.1 L2
3.5.2 D
3.5.3 S
3.5.4 Sinv
3.5.5 TTable

Usage:

```
#include <Level2/acquire1D.h>
const super_op& acquire1D::L2()
const gen_op& acquire1D::D()
const matrix acquire1D::S()
const matrix& acquire1D::Sinv()
const Ttable& acquire1D::TTable()
```

Description:

These functions allow users to obtain the internal variables of the ***acquire1D***. The function ***L2*** will return the working evolution Liouvillian. Function ***D*** returns the detection operator. Function ***S*** returns the eigenvector array of ***L2***. Function ***Sinv*** returns the inverse eigenvector array of ***L2***. And function ***TTable*** returns any existing transtions table.

Return Value: various

Example:

```
#include <acquire.h>
```

3.5.6 Detector

Usage:

```
#include <Level2/acquire1D.h>
void acquire1D::Detector(gen_op& D)
```

Description:

Thie function allows the user to change the acquisition detection operator to ***D***.

Return Value: void

Example:

```
#include <gamma.h>
main()
  {
  .....                                  // Generate an Acquire1D variable ACQ
  .....                                  // Generate new operator Ix1Iz2
  ACQ.Detector(Iz1Iz2);                  // Set detector now for Ix1Iz2
  }
```

3.6 Spectrum Generation Functions

3.6.1 T

3.6.2 Ts

Usage:

#include <Level2/acquire1D.h>

row_vector acquire1D::T(const gen_op& sigmap, int npts, double tinc);

void acquire1D::T(const gen_op& sigmap, row_vector& dataz, double tinc);

Description:

Function ***T*** generates a time domain spectrum associated with the acquisition. The points will be separated in time by ***tinc*** seconds. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned. The operator ***sigmap*** is taken to be the density operator representing the state of the system at the acquisition beginning.

Return Value: void or row_vector

Examples:

```
#include <gamma.h>
main ()
  {
  spin_system sys;                          // An isotropic spin system
  sys.read("system.sys");                   // Read in the spin system
  gen_op H = Ho(sys);                       // Isotropic NMR Hamiltonian
  gen_op D = Fm(sys);                       // Detection operator F-
  acquire1D Acq(H,D);                       // Set up for an acquisition
  gen_op sigmap = Fx(sys);                  // Take Fx as state as acquisition start
  double td = 1.e-3;                        // Take this as our dwell time
  int npts = 4096;                          // Take this as our block size
  row_vector fid = Acq.T(sigmap, npts,td); // Generate a time domain spectrum
  row_vector data(1000);                    // Another data vector
  Acq.T(sigmap,data,2.5*td);                // Fill data with another spectrum
  }
```

See Also: F, Fs, FD

3.6.3 F

3.6.4 Fs

Usage:

#include <Level2/acquire1D.h>

row_vector acquire1D::F(int npts, double Fst, double Ffi) const

void acquire1D::F(row_vector data, double Fst, double Ffi) const

Description:

Function ***F*** generates a frequency domain spectrum by summing over complex Lorentzians associated with each transition. The output will span the frequency range [***Fst, Ffi***] where the two frequencies are input in Hz. If a row_vector is input, ***data***, it will be completely filled. If a number of points, ***npts***, is specified a new vector of that size will be returned.

Return Value: void or row_vector

Example:

```
#include <gamma.h>
main ()
  {
  spin_system sys;                          // Declare a spin system
  sys.read("filein.sys");                   // Read in the spin system
  gen_op H = Ho(sys);                       // Isotropic NMR Hamiltonian
  gen_op D = F-(sys);                       // Detection operator F-
  acquire1D Acq(H,D);                       // Set up for an acquisition
  gen_op sigp = Fx(sys);                    // Take Fx as state as acquisition start
  double Fst = -1500.0;                     // Take this as start frequency
  double Ffi = 1500.0;                      // Take this as final frequency
  int npts = 4000;                          // Take this as our block size
  row_vector S=Acq.F(sigp,npts,Fst,Ffi);    // Generate frequency domain spectrum
  row_vector data(1000);                    // Another data vector
  Acq.D(sigmap,data,Fst, Ffi);              // Fill data with another spectrum
  }
```

See Also: T

3.6.5 FD

Usage:

```
#include <Level2/TrnsTable1D.h>
PulCycle TTable1D::CycTTable1D1(const spin_system& sys)
```

Description:

The ***TTable1D*** member function ***CycTTable1D1*** returns a pulse cycle using the 25 step TTable1D-1 pulse sequence coupled to a WALTZ-4 cycle.

Return Value:

A TTable1D-1 pulse cycle is returned.

Example:

```
#include <gamma.h>
main ()
  {
  spin_system sys;                          // Declare a spin system
  sys.read("filein.sys");                   // Read in the spin system
  TTable1D TT;                              // Declare TTable1D Parameters
  TT.read("filein.pset")                    // Read in TTable1D Parameters
  PulCycle GCy = TT.CycTTable1D1(sys); // TTable1D-1 pulse cycle
```

See Also: WF, PCmp

3.6.6 FD

Usage:

```
#include <Level2/TrnsTable1D.h>
PulCycle TTable1D::CycTTable1D1(const spin_system& sys)
```

Description:

The ***TTable1D*** member function ***CycTTable1D1*** returns a pulse cycle using the 25 step TTable1D-1 pulse sequence coupled to a WALTZ-4 cycle.

Return Value:

A TTable1D-1 pulse cycle is returned.

Example:

```
#include <gamma.h>
main ()
  {
  spin_system sys;                          // Declare a spin system
  sys.read("filein.sys");                   // Read in the spin system
  TTable1D TT;                              // Declare TTable1D Parameters
  TT.read("filein.pset")                    // Read in TTable1D Parameters
  PulCycle GCy = TT.CycTTable1D1(sys); // TTable1D-1 pulse cycle
```

See Also: WF, PCmp

3.7 Auxiliary Functions

3.7.1 ls

Usage:

```
#include <Level2/acquire1D.h>
int acquire1D::ls()
int acquire1D::size()
```

Description:

The function ***ls*** returns the ***acquire1D*** Liouville space dimension, and will be the square of the Hilbert space dimension. The funciton ***size*** returns the value of ***pos***, the number of non-zero components that make up the acquisition. The funciton full_size

Return Value: integer

Example:

```
#include <acquire.h>
```

See Also:

3.7.2 Output Functions

3.7.3 print

3.7.4 <<

Usage:

#include <Level2/acquire1D.h>
ostream& acquire1D::print(ostream& ostr) const
ostream& acquire1D::print(ostream& ostr, gen_op& sigmap) const
ostream& operator << (ostream& ostr, acquire1D& ACQ)

Description:

These functions all send ASCII formatted output into the supplied output stream ***ostr***. If the ***print*** function is used with only ***ostr*** as an argument then the generic core elements of ***acquire1D*** are placed into the output stream. If, in addition, a prepared density operator is supplied, ***sigmap***, then components of the acquisition are sent to the output stream. The operator **<<** adds the acquisition specified as an argument ***ACQ*** to the output stream ***ostr***:

non-zero points out of # possible
Dwell time: # (if available)
A[i], B[i] pairs
Hilbert space basis.

Return Value: ostream

Example(s):

#include <acquire.h>

See Also:

3.7.5 write

Usage:

#include <Level2/acquire1D.h>
void acquire1D::write(const string& filename) const
ofstream& acquire1D::write(ofstream& fp) const

Description:

The function ***write*** is used for binary output of an ***acquire1D***. Given a ***filename*** as input, print will create the file and write the acquisiton into it. Given an output file stream, ***fp***, the function will write the ***acquire1D*** into the filestream at its current position.

Return Value: void

Example:

#include <acquire.h>

See Also:

3.7.6 read

Usage:

#include <Level2/acquire1D.h>
void acquire1D::read(const string& filename)
ofstream& acquire1D::read(ifstream& fp)

Description:

The function ***read*** is used for binary input of an ***acquire1D***. Given a ***filename*** as input, print will open the file and read the acquisiton from it. Given an input file stream, ***fp***, the function will red the ***acquire1D*** from the filestream at its current position.

Return Value: void

Example:

#include <acquire.h>

See Also: write

3.8 Description

3.8.1 Introduction

Class acquire1D contains the computational machinery for the determination of an expectation values in a system which has undergone a single time evolution[1]. This case arises in problems where the system is initially prepared into some state represented by the density operator $|\sigma_{prep}>$, allowed to evolve for some time t, during which expectation values are sampled.

The expectation values are given by

$$\langle Op(t)\rangle \;=\; Tr\{Op\cdot\sigma(t)\} \;=\; \langle Op^{\dagger}|\sigma(t)\rangle \tag{3-1}$$

where the physical quantity desired corresponds to the operator ***Op***. Class ***Acquire1D*** provides a simple means of calculating such values with a high degree of computational efficiency. Functions are provided which produce arrays of these values in *both the time and frequency domains*.

3.8.2 Hilbert Space Acquisition

Consider the situation where the system is evolving under a constant static Hamiltonian, $\boldsymbol{H}$. The density operator representing the system changes in time as given by the solution to the Liouville equation[2]

$$\sigma(t) \;=\; e^{-i\boldsymbol{H}t}\sigma_p e^{i\boldsymbol{H}t} \;=\; U_t\sigma_p U_t^{-1} \;=\; U_t\sigma_p U_t^{\dagger} \tag{3-2}$$

where σ_p represents the prepared system at the evolution start. We can immediately write a equation for how the expectation values change in time.

$$\langle Op(t)\rangle \;=\; Tr\left\{Op\cdot e^{-i\boldsymbol{H}t}\sigma_p e^{i\boldsymbol{H}t}\right\} \;=\; Tr\left\{Op\cdot U_t\sigma_p U_t^{-1}\right\} \;=\; Tr\{Op\cdot U_t\sigma_p U_t^{\dagger}\}$$

Now, the problem approached by class ***acquire1D*** is when one wishes to monitor the expectation value over a series of time increments. Using t_k for the k^{th} point (which occurs at time t from t_o at σ_p), we have from the previous equation

$$\langle Op(t_k)\rangle \;=\; Tr\{Op\sigma(t_k)\} \;=\; Tr\left\{Op\cdot e^{-i\boldsymbol{H}t_k}\sigma_p e^{i\boldsymbol{H}t_k}\right\}.$$

If the points themselves are equally spaced in time then we know that $t_k = k\Delta t$ where Δt is the time increment. Switching to propagator notation with U for this incremental time evolution, $U = e^{-iH\Delta t}$

1. It is assumed that the system is described by the density operator σ and that any evolution in time will be described by a solution to the Liouville equation.
2. If Hamiltonian $\boldsymbol{H}$ is Hermitian, then the propagator U_t is unitary and its inverse equals its adjoint, $U_t^{-1} = U_t^{\dagger}$

our equation becomes

$$\langle Op(t_k)\rangle \;=\; Tr\left\{ Op\cdot \boldsymbol{U}^k \sigma_p [\boldsymbol{U}^{-1}]^k \right\} \;=\; Tr\left\{ Op\cdot \boldsymbol{U}^k \sigma_p U_t\dagger^k \right\}. \tag{3-3}$$

Expanding the trace relationship as matrix multiplications, where ***hs*** is the Hilbert space dimension yeilds

$$\langle Op(t_k)\rangle \;=\; \sum_{\alpha}^{hs}\sum_{\alpha'}^{hs}\sum_{\beta}^{hs}\sum_{\beta'}^{hs} \langle\alpha|Op|\alpha'\rangle\langle\alpha'|\boldsymbol{U}^k|\beta\rangle\langle\beta|\sigma_p|\beta'\rangle\langle\beta'|U_t\dagger^k|\alpha\rangle.$$

If we work in the eigenbasis of ***H***, the propagator ***U*** is represented by a diagonal matrix. Then the only non-zero contributions to the summation(s) occur when α'=β and β'=α. Our working thus simplifies in the H eigenbasis as

$$\langle Op(t_k)\rangle \;=\; \sum_{\alpha}^{hs}\sum_{\alpha'}^{hs} \langle\alpha|Op|\alpha'\rangle\langle\alpha'|\boldsymbol{U}^k|\alpha'\rangle\langle\alpha'|\sigma_p|\alpha\rangle\langle\alpha|U_t\dagger^k|\alpha\rangle.$$

Keep in mind the three conditions we have used so far. First, the expectation value points are equally spaced and separated by the a constant time increment (taken up in the propagator ***U***). Second, then density matrix is evolving under the effects of a time independent Hamiltonian, ***H***. Third, all operators are expressed in the eigenbasis of this Hamiltonian.

There are two more things worth considering with regards to the previous equation. Note that the propagators are merely complex exponentials which are oscillating at the eigenvalues (frequencies) of the Hamiltonian. The two propagator terms place an oscillation on every element of the density matrix at their individual associated frequencies. This is the normal effect of a static Hamiltonian on the density matrix. Also of note is that the operator for the physical quantity typically acts as a filter to select out the density matrix elements of importance. This implies that only a limited number of elements in the double summation actually contribute to the calculated points (due to the typically sparse nature of the operator ***Op***).

We can take advantage of this latter fact. Reordering the elements of our previous equation,

$$\langle Op(t_k)\rangle \;=\; \sum_{\alpha}^{hs}\sum_{\alpha'}^{hs} \langle\alpha|Op|\alpha'\rangle\langle\alpha'|\sigma_p|\alpha\rangle\langle\alpha'|\boldsymbol{U}^k|\alpha'\rangle\langle\alpha|U_t\dagger^k|\alpha\rangle,$$

we can simply create a new index, αα' and form a vector ***B*** from the propagator elements.

$$B_{\alpha\alpha'} \;=\; \langle\alpha'|\boldsymbol{U}|\alpha'\rangle\langle\alpha|U_t\dagger|\alpha\rangle \;=\; \langle\alpha'|\boldsymbol{U}|\alpha'\rangle\langle\alpha|U_t|\alpha\rangle \tag{3-4}$$

Similarly, we form a second new vector ***A*** from the operator ***Op*** and a third new vector ***C*** for the initial density operator[1], σ_p.

$$A_{\alpha\alpha'} = \langle\alpha|Op|\alpha'\rangle \qquad C_{\alpha'\alpha} = \langle\alpha'|\sigma_p|\alpha\rangle = \langle\alpha|\sigma_p{}^*|\alpha'\rangle = C_{\alpha\alpha'}{}^* \tag{3-5}$$

Our equation now is now quite compact,

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls} A_{\alpha\alpha'}[B_{\alpha\alpha'}]^k C_{\alpha'\alpha} = \sum_{\alpha\alpha'}^{ls} A_{\alpha\alpha'}[B_{\alpha\alpha'}]^k C_{\alpha\alpha'}{}^*. \tag{3-6}$$

We have used ***ls*** to represent the Liouville space dimension, and this relates to the Hilbert space dimension by

$$ls = hs \cdot hs. \tag{3-7}$$

Equation (3-7) is not the final form which is used by class ***acquire1D***. We have now to switch the sum to include only the terms for which the <α|***Op***|α'> are non-zero (or equivalently those for which $A_{\alpha\alpha'}$ will be non-zero), typically only a select few[2] in the sum over all α and α'.

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls} A_{\alpha\alpha'}[B_{\alpha\alpha'}]^k C_{\alpha\alpha'}{}^* = \sum_{p}^{\substack{Op_{\alpha\alpha'}\neq 0 \\ \leq ls""}} A_p[B_p]^k C_p{}^* \tag{3-8}$$

In this formulation, the vector ***A*** contains part of the intensity information and the vector ***B*** contains all the frequency information. Both need to be computed only once no matter how many expectation value points are desired. The sum over ***p*** is repeated for each ***k*** (time), but ***p*** is usually much smaller than the original equation form which contained a full sum over α and α' which each spanned the entire Hilbert space of the system.

This is the scheme which GAMMA takes advantage of through the use of the class ***acquire1D***. Given an operator ***Op*** and a propagator[3] for a single time increment ***U***, class ***acquire1D*** formulates the vector ***B*** and determines the minimum number of elements in the sum involving the index ***p***. It also produces a prototype vector ***A*** which contains relevant information about ***Op***. Subsequently, when supplied with an initial density matrix $\sigma(t_o)$, the vector ***A*** is fully formed and the expectation values determined according to equation (3-8). The vector ***A*** can be rebuilt with any number of new $\sigma(t_o)$ matrices and more expectation values generated.

1. The density operator is by definition Hermitian.
2. The number of non-zero elements depends upon the operator. For NMR simulations this is often ***F_*** which is typically represented by a sparse array.
3. Functions are supplied that accept the equivalent of ***U***, the static Hamiltonian ***H*** and the increment time Δ***t***

3.8.3 Liouville Space Acquisition

Often, the equation of motion for the density matrix is written using superoperators in Liouville space rather than in terms of Hilbert space operators. We have seen that under the effects of a constant static Hamiltonian, $\boldsymbol{H}$, the density matrix changes in time as given by the solution to the Liouville equation, equation (3-2).

$$\sigma(t) = e^{-i\boldsymbol{H}t}\sigma(t_o)e^{i\boldsymbol{H}t} = \boldsymbol{U}\sigma(t_o)\boldsymbol{U}^{-1}$$

If we utilize the unitary transformation superoperator[1] equivalent of $\boldsymbol{U}$, $\Gamma_{\boldsymbol{U}}$, this equation takes the form

$$|\sigma(t)\rangle = \Gamma_U|\sigma(t_o)\rangle \tag{3-1}$$

where now both the superoperator $\Gamma_{\boldsymbol{U}}$ and the ket $|\sigma\rangle$ reside in Liouville space, the latter being the equivalent to the Hilbert space operator σ. Equation (3-1) can always be written when the system evolves under a static Hamiltonian, however there are other instances when an equation of this form is applicable yet having no equivalent Hilbert space formulation. Thus we shall now treat the generalized form where Γ is some generic evolution superoperator.

$$|\sigma(t)\rangle = \Gamma|\sigma(t_o)\rangle \tag{3-2}$$

The expectation value equation, analogous to equation (3-3), is

$$\langle Op(t_k)\rangle = Tr\{Op\cdot\Gamma^k\sigma(t_o)\}.$$

The trace in superoperator notation is given by[2]

$$Tr\{AB\} = \langle A^\dagger|B\rangle = (|A^\dagger\rangle, |B\rangle). \tag{3-3}$$

where $|\boldsymbol{A}\rangle$ is now a "ket" representing the matrix $\boldsymbol{A}$ and $(|A^\dagger\rangle, |B\rangle)$ the scalar product between the adjoint of $|\boldsymbol{A}\rangle$ and $|\boldsymbol{B}\rangle$. Using this notation we have

$$\langle Op(t_k)\rangle = Tr\{Op\cdot\Gamma^k\sigma(t_o)\} = \langle(Op)^\dagger|\Gamma^k|\sigma(t_o)\rangle \tag{3-4}$$

and expansion of the scalar product then produces

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls}\langle 1|Op^\dagger|\alpha\alpha'\rangle[\langle\alpha\alpha'|\Gamma|\alpha\alpha'\rangle]^k\langle\alpha\alpha'|\sigma(t_o)|1\rangle, \tag{3-5}$$

which is obviously similar to the previous Hilbert space formulation. In the Hilbert space treatment in which it was important to work in the eigenbasis of the propagator $\boldsymbol{U}$. Now we must work in the

1. See GAMMA User Documentation on ***class super_op*** for a discussion of such superoperators.
2. See "Superoperators in Magnetic Resonance" by J. Jeener, *Adv. Magn. Reson*, **10**, 1 (1982).

eigenbasis of Γ (which is in Liouville space). Using the relationship $\Gamma = \boldsymbol{S}\Lambda\boldsymbol{S}^{-1}$ where Λ is a diagonal matrix containing the eigenvalues of Γ and $\boldsymbol{S}$ a matrix containing its eigenvalues, we must expand the previous equation a bit.

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls} \langle 1|Op^\dagger|\alpha\alpha'\rangle\langle\alpha\alpha'|S\Lambda^k S^{-1}|\alpha\alpha'\rangle\langle\alpha\alpha'|\sigma(t_o)|1\rangle$$

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls}\sum_{\beta\beta'}^{ls} \langle 1|Op^\dagger|\alpha\alpha'\rangle[\langle\alpha\alpha'|S|\beta\beta'\rangle\langle\beta\beta'|\Lambda^k|\beta\beta'\rangle\langle\beta\beta'|S^{-1}|\alpha\alpha'\rangle]\langle\alpha\alpha'|\sigma(t_o)|1\rangle$$

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls} \langle 1|Op^\dagger S|\alpha\alpha'\rangle\langle\alpha\alpha'|\Lambda^k|\alpha\alpha'\rangle\langle\alpha\alpha'|S^{-1}\sigma(t_o)|1\rangle$$

Letting (analogous to (3-4) & (3-5))

$$B_{\alpha\alpha'} = \langle\alpha\alpha'|\Lambda|\alpha\alpha'\rangle \qquad A_{\alpha\alpha'} = \langle 1|Op^\dagger S|\alpha\alpha'\rangle\langle\alpha\alpha'|S^{-1}\sigma(t_o)|1\rangle \tag{3-6}$$

we have once again obtain equation (3-8)

$$\langle Op(t_k)\rangle = \sum_{\alpha\alpha'}^{ls} A_{\alpha\alpha'}[B_{\alpha\alpha'}]^k = \sum_{p}^{\substack{\langle 1|Op^\dagger S|\alpha\alpha'\rangle \neq 0 \\ \leq ls""}} A_p[B_p]^k$$

We again restrict the summation to be less than the Liouville space dimension although now the basis for choosing when the elements of $\boldsymbol{A_p}$ will be zero are changed to when $\langle\boldsymbol{1}|\boldsymbol{Op}\dagger\boldsymbol{S}|\alpha\alpha'\rangle$ is zero rather than when $\langle\alpha|\boldsymbol{Op}|\alpha'\rangle$ is zero.

It should be noted that when the superoperator Γ is the unitary transformation superoperator Γ_U, then it will be diagonal when the operator $\boldsymbol{U}$ is in its eigenbasis. This is clearly seen by the definition of the unitary transformation superoperator,

$$\Gamma_U = \boldsymbol{U} \otimes \boldsymbol{U}^*$$

In that instance the matrix $\boldsymbol{S}$ is the identity matrix and the element $\langle\boldsymbol{1}|\boldsymbol{Op}\dagger\boldsymbol{S}|\alpha\alpha'\rangle$ becomes $\langle\boldsymbol{1}|\boldsymbol{Op}\dagger\boldsymbol{S}|\alpha\alpha'\rangle = \langle\alpha|\boldsymbol{Op}|\alpha'\rangle$. The result is that the two treatments presented, that in the Hilbert space and that in the Liouville space, produce the identical result.

0.0.1 BWR Relaxation

We next consider the situation where the system is evolving under the effects of relaxation in the liquid state as treated by Redfield theory. The equation describing system evolution takes the form[1]

$$|\sigma(t)\rangle = e^{-iLt}|\sigma(t_o) - \sigma_{inf}\rangle + |\sigma_{inf}\rangle$$

In this case the superoperator e^{-iLt} will take any density matrix to zero at infinite time, and the density operator σ_{inf} is the state of the system at infinite time (normally equilibrium or some steady-state). The expectation value equation becomes a bit messy relative to the previous treatments. We have then

$$\langle Op(t_k)\rangle = Tr\left\{ Op \cdot \left[e^{-iLt_k}(\sigma(t_o) - \sigma_{inf}) + \sigma_{inf} \right] \right\}$$

which in superoperator notation is given by

$$\langle Op(t_k)\rangle = \langle Op^\dagger| \left[e^{-iLt_k}|\sigma(t_o) - \sigma_{inf}\rangle + |\sigma_{inf}\rangle \right] \tag{3-7}$$

Both the trace operation and the scalar product are linear so we have

$$\langle Op(t_k)\rangle = \langle Op^\dagger|e^{-iLt_k}|\sigma(t_o) - \sigma_{inf}\rangle + \langle Op^\dagger|\sigma_{inf}\rangle \tag{3-8}$$

Expansion of the scalar product produces

$$\langle Op(t_k)\rangle = \langle Op^\dagger|\sigma_{inf}\rangle + \sum_{\alpha\alpha'}^{ls} \langle 1|Op^\dagger|\alpha\alpha'\rangle [\langle \alpha\alpha'|e^{-iL\Delta t}|\alpha\alpha'\rangle]^k \langle \alpha\alpha'|\sigma(t_o) - \sigma_{inf}|1\rangle$$

Because the value of $\langle Op^\dagger|\sigma_{inf}\rangle$ is constant for all values of t_k we do not include it in the expansion. Letting $\Gamma = e^{-iL\Delta t}$, $\sigma'(t_o) = \sigma(t_o) - \sigma_{inf}$, and subtracting the constant trace from both sides we obtain

$$\langle Qp(t_k)\rangle = \langle Op(t_k)\rangle - \langle Op^\dagger|\sigma_{inf}\rangle = \sum_{\alpha\alpha'}^{ls} \langle 1|Op^\dagger|\alpha\alpha'\rangle [\langle \alpha\alpha'|\Gamma|\alpha\alpha'\rangle]^k \langle \alpha\alpha'|\sigma'(t_o)|1\rangle$$

1. GAMMA can place Redfield relaxation in the context of a relaxation propagator. The two treatments are related as $\sigma(t) = \Gamma\{\sigma(t_o) - \sigma_{inf}\} + \sigma_{inf} = \Gamma_{RP}\sigma(t_o)$. The treatment using relaxation propagators fits into class acquire as specified in the previous section. (See the GAMMA documentation on relaxation effects in liquids.)

By comparison with equation (3-5), we see that we have already evaluated $\langle Op(t_k)\rangle$.

$$\langle Op(t_k)\rangle = \sum_{\substack{\langle 1|Op^\dagger S|\alpha\alpha'\rangle \neq 0 \\ \leq ls""}} A_p[B_p]^k$$

Thus the true expectation value is given by

$$\langle Op(t_k)\rangle = \langle Op^\dagger|\sigma_{inf}\rangle + \sum_{\substack{\langle 1|Op^\dagger S|\alpha\alpha'\rangle \neq 0 \\ \leq ls""}} A_p[B_p]^k \qquad (3\text{-}9)$$

For clarity, we shall explicitly write down the elements of the two vectors, analogous to (3-6),

$$B_{\alpha\alpha'} = \langle\alpha\alpha'|\Lambda|\alpha\alpha'\rangle \qquad A_{\alpha\alpha'} = \langle 1|Op^\dagger S|\alpha\alpha'\rangle\langle\alpha\alpha'|S^{-1}[\sigma(t_o)-\sigma_{inf}]|1\rangle \qquad (3\text{-}10)$$

an note that in this case $\Gamma = \boldsymbol{e^{-iL\Delta t} = S\Lambda S^{-1}}$.

There is a final point worth mentioning with regards to this Redfield approach. Often, one will be looking for an expectation value which is zero a infinite time. If true, than the constant term in equation (3-9) is zero

$$\langle Op^\dagger|\sigma_{inf}\rangle = Tr\{Op\cdot\sigma_{inf}\} = 0 \qquad (3\text{-}11)$$

and can be ignored. As a consequence, equation (3-8) simplifies to

$$\langle Op(t_k)\rangle = \langle Op^\dagger|e^{-iLt_k}|\sigma(t_o)-\sigma_{inf}\rangle = \langle Op^\dagger|e^{-iLt_k}|\sigma(t_o)\rangle - \langle Op^\dagger|e^{-iLt_k}|\sigma_{inf}\rangle.$$

In the Redfield treatment, $\boldsymbol{e^{-iLt_k}}|\sigma_{inf}\rangle$ will be different than $|\sigma_{inf}\rangle$. It turns out that when the trace relationship in equation (3-11) is satisfield, it is often true that

$$Tr\left\{Op\cdot e^{-iLt_k}\sigma inf\right\} = 0 \qquad (3\text{-}12)$$

as well because the random relaxation processes acting on $|\sigma_{inf}\rangle$ will usually not generate contributions to the expectation value if $|\sigma_{inf}\rangle$ contributes nothing to it . Having both (3-11) and (3-12) satisified causes equation (3-8) to be equal to

$$\langle Op(t_k)\rangle = \langle Op^\dagger|e^{-iLt_k}|\sigma(t_o)\rangle,$$

and this has the same form as the trace relationship in the previous section, *i.e.* equation (3-4). The end result is that it will then make no difference whether you perform an acquisition using equa-

tions of the previous section which completely ignore $|\sigma_{inf}>$ or use the equations here.

0.0.2 Time Domain Spectra

The array of expectation values which may be formed using ***acquire1D*** can be thought of as a 1-dimensional spectrum in the time domain. An efficient formula for such a spectrum is obtained by substitution of the quantities defined in (3-13) into (3-13).

$$S(t) = \sum_{i}^{N \le ls} A_i e^{B_i t} C_i + K \tag{3-13}$$

Note that the spectrum consists of N complex exponentials, each oscillating about the the constant K. The intensities of the exponentials are specified by the complex number A_iC_i (from which one may obtain the magnitude and phase) and the complex value B_i contains the ocillation frequencies and decay rates.

A modification of this is useful when the times at which the spectrum is evaluated are evenly spaced, i.e. $t = n \times \Delta t$ where n is an integer. In such an event we can replace Eq. (3-13) with

$$S(n\Delta t) = \sum_{i}^{N \le ls} EXB_i^n A_i C_i + K \tag{3-14}$$

where

$$EXB_i^n = [e^{B_i \Delta t}]^n = e^{B_i n \Delta t} = e^{B_i t} \qquad n = 0, 1, 2, \ldots, N-1 \tag{3-15}$$

3.8.4 Frequency Domain Spectra

An equation for frequency domain spectra can be generated from the Fourier transform (FT) of the time domain spectrum, (3-13). There is a slight conceptual problem in that the Fourier transform extends in time to span ***(-∞, ∞)***, whereas experimental data acquired in time will normally span ***[0, t]*** and the frequency spectrum obtained from a discrete Fourier transform (DFT). These differences can be inconsequential under the appropriate conditions, but we must keep in mind that a DFT on digitized points does not ***exactly*** match the analog FT performed here.

It will be assumed either that the observed frequency spectrum will be the result of a FT of the data acquired from time 0 to infinite time. The Fourier integrals and a discussions of differences between treatments will be left for a later discussion.

$$S(\omega) = \int_0^{\infty} S(t) e^{-i\omega t} dt = \int_0^{\infty} \left\{ \sum_i^{N \le ls} e^{B_i t} A_i C_i + K \right\} e^{-i\omega t} dt \tag{3-1}$$

Expansion of this produces the rather tedious equation

$$S(\omega) = \sum_i^{N \le ls} A_i C_i \int_0^{\infty} e^{B_i t} e^{-i\omega t} dt + K \int_0^{\infty} e^{-i\omega t} dt \tag{3-2}$$

and we must now evaluate the integrals. They shall not be formally evaluated here[1], instead we will borrow a solution[2].

$$\int_0^{\infty} e^{-\alpha t} e^{-i\omega t} dt = \frac{\alpha - i\omega}{\alpha^2 + \omega^2} = \frac{1}{\sqrt{\alpha^2 + \omega^2}} e^{i \operatorname{atan}(-\omega/\alpha)} \qquad \alpha = real \tag{3-3}$$

The integrals in our frequency spectrum equation are evaluated as follows

$$\int_0^{\infty} e^{-i\omega t} dt = \frac{-i}{\omega} \qquad \int_0^{\infty} e^{B_i t} e^{-i\omega t} dt = \frac{r_i - i(\omega - \omega_i)}{r_i^2 + (\omega - \omega_i)^2} \tag{3-4}$$

1. The integral ranges are extended to **-∞** using the Heavyside function, thus becoming Fourier transforms. They are then evaluated using the similarity, shift, and convolution theorems. See "The Fourier Transform and It's Applications", R.N. Bracewell, 2nd. Edition, McGraw-Hill Book Company, New York, 1978.
2. See "The Fast Fourier Transform", E.O. Brigham, Prentice-Hall, Inc., New Jersey, 1974. The integral can be found on page 12 of this text.

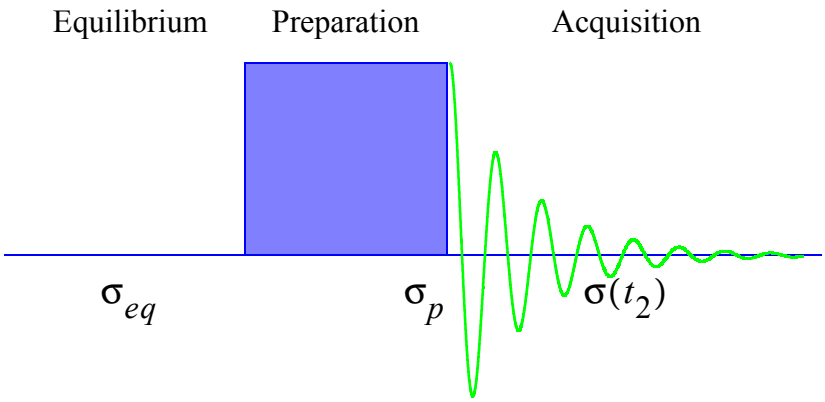
which we can write in a shorthand notation as

$$S(\omega) = \sum_{i}^{N \le ls} \frac{r_i - i(\omega - \omega_i)}{r_i^2 + (\omega - \omega_i)^2} + K\left[\delta(\omega) - \frac{i}{\omega}\right] \tag{3-5}$$

using the following definitions.

3.8.5 Application to NMR

There are many 1-dimensional nuclear magnetic resonance (NMR) experiments based on the preparation-evolve sequence described in the previous section. These experiments can be schematically represented by the following pulse sequence diagrams.



Although NMR experiments which do not fit this format may be treated with GAMMA, class ***acquire1D*** cannot be directly applied to them.

For ***acquire1D*** we must specify 1.) What we wish to detect, 2.) How the system evolves, and 3.) Our prepared density operator - the acquisition starting point, σ_p.

3.8.5.1 The Detection Operator

The goal in NMR is to detect transverse magnetization and examine its frequency components. We know from first principles that any magnetization will experience a torque in a static field,

$$\frac{d}{dt}\vec{M} \propto \vec{M} \times \vec{B}$$

and by simple application of the right-hand rule (where the static field B points along the +*z* axis in GAMMA) that magnetization in the transverse plane will precess in ***clockwise*** fashion looking down from +*z* into the ***xy-plane***. 1

Consider a spin having chemical shift of ω. It's associated shift Hamiltonian will be given by $\boldsymbol{H}_{cs} = -\omega \boldsymbol{I}_z$ assuming a positive gyromagnetic ratio and GAMMA's having it's static field along +*z*. Transverse magnetization associated with this frequency will rotate in ***clockwise*** fashion looking down from +*z* into the ***xy-plane*** .

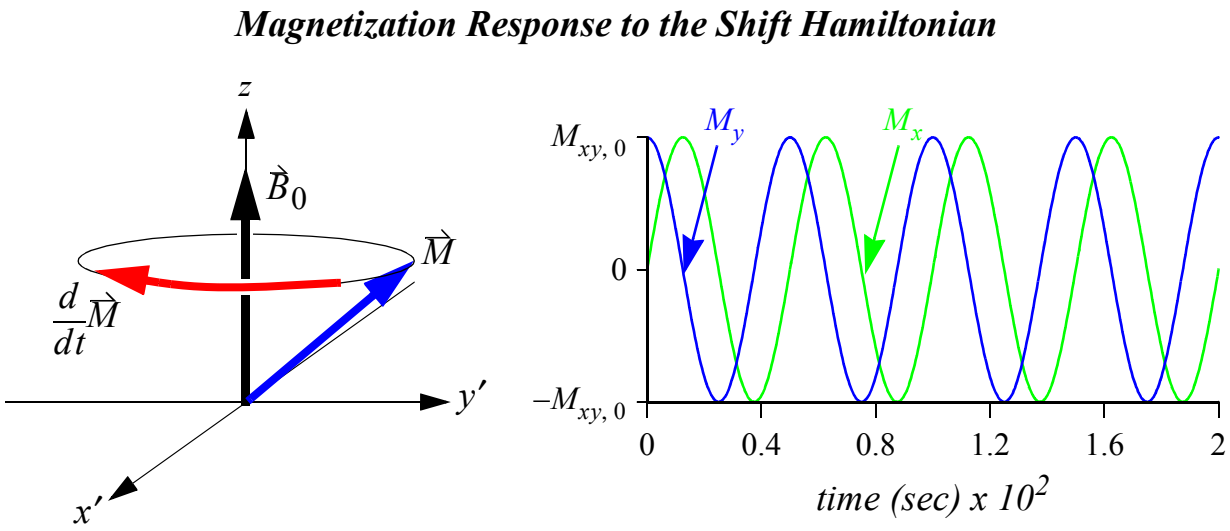


Figure 4-1 - Evolution of magnetization under the effects of the shift Hamiltonian. The figures are shown are for a single spin having a chemical shift of 200 Hz relative to a rotating frame following a pulse applied along the x' axis of 45 degrees. The figure on the left is constructed from classical arguments based on the cross product between $\vec{M}$ and $\vec{B}_0$. The figure on the right shows magnetization components of a single proton evolving in time in the rotating frame.

Normally ***the detection operator in NMR will be $F_- = F_x - iF_y$.*** This implies that since we measure transverse magnetization in quadrature[1] mode. WThis tracks magnetization as complex number, reals along the x-axis (proprotional to F_x) and imaginaries along the negative y-axis (proprotional to $-F_x$). This does not preclude users from choosing a different "detection" operator. For example, simply F_x if quadrature detection is not required. Or perhaps on wishes to follow a specific operator, say $I_{z1}I_{x2}$, which can be done computationally even though it does not correspond to a directly measurable quantity in the experiment.

3.8.5.2 System Evolution

The simplest method of evolving the system would be under a static Hamiltonian in Hilbert space. This would include the isotropic high-resolution NMR Hamiltonian used for liquids and a modification of it in the rotating frame when CW irradiation (e.g. decoupling) is applied during the acquisiton. Such treatments ignore the effects of relaxation and exchange.

1. This is quadrature detection in the F2/T2 domain. The Fourier transform (FT) of a pure sine or cosine is unable to distinguish positive and negative frequencies, but the FT on a complex exponential will distinguish them. An artifact of using a discrete Fourier transform (DFT) on data spanning $[0,\infty)$ rather than a full transform over all time is that the DFT will always produce a spectrum of real data and a derivative of this same spectrum as imaginary data. The two sets of data after the DFT are redundant, but useful in allowing phase manipulations of frequency domain data.

3.8.6 Class Acquire Equations

In this section we regroup the applicable equations regarding ***class acquire***, or equivalent regarding repetitive computation of expectation values.

Class Acquire Equations

Expectation Value at Time t_k

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

Unitary Transformation, Hilbert Space

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

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

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

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

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

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

Non-Unitary Transformation, Liouville Space

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

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

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

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

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

Redfield Theory, Liouville Space

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

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

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

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

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

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

3.8.7 Internal Structure

The internal structure of class ***acquire1D*** is relatively simple as it is based on the simple formula.

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

However, mostly because class ***acquire1D*** supports both Hilbert space and Liouville space computations, the structure contains additional entities beyond those explicit in this equation. These are given in the following table.

Internal Composition of Class Acquire

Declaration	Variable	Definition
int LS;	***ls***	Liouville space of the system, ***ls = hs*hs***
int pos;	***pos***	Range of the index ***p***: p∈ ***[0,pos]***. Always, ***pos*** ≤ ***ls***
complex* A;	$\Rightarrow \boldsymbol{A_p}$	Pre-assembled form of vector $\boldsymbol{A_p}$ (detector equivalent)
complex* B;	$\boldsymbol{B_p}$	Fully assembled vector $\boldsymbol{B_p}$ (time propagator equivalent)
int *I;	--------	Indexing array, relates index ***p*** to the index α or αα′
double DCUTOFF	--------	Detection cutoff limit. Sets where *noise* level is.
super_op L;	***L***	System Liouvillian (if Liouville space calculation)
matrix Sm1	$\boldsymbol{S^{-1}}$	Inverse superoperator eigenvector matrix
gen_op det	***Op***	Detection operator (in Hilbert space)
gen_op siginf;	σ_{inf}	Infinite time density operator (normally σ_{eq} or σ_{ss})
complex trinf;	$\langle Op\|\sigma_{inf}\rangle$	Trace at infinite time, a constant often zero.
TTable1D TTab	--------	A table of transitions (generates FID's and spectra)
gen_op sigmap;	σ_p	Initial (prepared) density operator, at acquisition start
double ICUTOFF	--------	Intensity cutoff limit. Sets where *noise* level is.

It is important to note that ***acquire1D*** does not store the vector $\boldsymbol{A_p}$, but a "prevector" which goes into the formation of the true $\boldsymbol{A_p}$ when the acquisition is supplied with a density matrix. The construction of $\boldsymbol{A_p}$ depends upon which treatment the particular acquisition is using. There are three treatments supported in class ***acquire1D***, as discussed in the previous Sections of this Chapter, one

for each means by which the density matrix is propagated in time.We will refer to these as Hilbert space, Liouville space, and Redfield[1]. (although the Redfield treatment is in Liouville space as well).

Class Acquire1D Hilbert Space Treatment

Variable	Definition	Origin and Value
LS	Liouville Space	
pos	Total Elements	Number of $\langle\alpha\|Op\|\alpha'\rangle \geq$ ***DCUTOFF***, ***pos*** $\leq$ ***hs*hs***
A	Detection Vector	From Op, $\langle\alpha\|Op\|\alpha'\rangle \geq$ ***DCUTOFF*** $\Rightarrow$ $\langle p\|A\rangle$
B	Evolution Array	H,U
I	I	con
I	I	con
det	Detection Operator	Set in construction, or by member function
trinf	Infinite Time Trace	Zero, nothing need in Hilbert space treatment
Sm1	Inverse Eigenvectors	Empty, no Liouvillian eigenvectors

Figure A The keyword *con* implies the value is set during Acquire1D construction. Keyword *fun* implies there is a function which can directly set this value. Keyword *def* implies there is a default value for this.

When the value of ***ls*** is zero, the acquisition is using a Hilbert space treatment. As such, elements of the vector $\boldsymbol{A_p}$ well be re-indexed elements from formula (3-5).

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

The elements of ***A*** in this case correspond to elements of $\langle\alpha|\boldsymbol{Op}|\alpha'\rangle$, but those which are zero are left out. The integer arrays ***I*** and ***J*** are used to relate the indices α and α' to the index ***p***. Supplied with the initial density operator σ_p, the vector $\boldsymbol{A_p}$ is easily built from ***A***, ***I*** and ***J***. The matrix ***Sm1*** is not utilized and remains unfilled.

1. Note that the "Redfield" treatment is also a Liouville space treatment involving the use of superoperators.

.

Table 1: Class Acquire Treatments

Type	complex* A	int* I	matrix Sm1	int ls
H	$A_p \Leftarrow \langle\alpha'\|\boldsymbol{U}\|\alpha'\rangle$	$p = \alpha\alpha' \forall \langle\alpha\|Op\|\alpha'\rangle \neq 0$	NULL	0
L	$A_p \Leftarrow \langle 1\|Op^\dagger S\|\alpha\alpha'\rangle$	$p = \alpha\alpha' \forall \langle 1\|Op^\dagger S\|\alpha\alpha'\rangle \neq 0$	S^{-1}	0
R	$A_p \Leftarrow \langle 1\|Op^\dagger S\|\alpha\alpha'\rangle$	$p = \alpha\alpha' \forall \langle 1\|Op^\dagger S\|\alpha\alpha'\rangle \neq 0$	S^{-1}	ls
	complex* B	int* J	matrix siginf	trinf
H	$B_p \Leftarrow \langle\alpha'\|\boldsymbol{U}\|\alpha'\rangle\langle\alpha\|\boldsymbol{U}^{-1}\|\alpha\rangle$	$p = \alpha\alpha' \forall \langle\alpha\|Op\|\alpha'\rangle \neq 0$	NULL	0
L	$B_p \Leftarrow \langle\alpha\alpha'\|\Lambda\|\alpha\alpha'\rangle$	NULL	NULL	0
R	$B_p \Leftarrow \langle\alpha\alpha'\|\Lambda\|\alpha\alpha'\rangle$	NULL	σ_{inf}	$\langle Op^\dagger\|\hat{\sigma}_{inf}\rangle$

When the value of ***ls*** is non-zero (it will be the Liouville space dimension) and the matrix ***siginf*** is empty, the acquisition is using a Liouville space treatment. As such, elements of the vector $\boldsymbol{A_p}$ well be re-indexed elements from formula (3-6).

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

The elements of $\boldsymbol{A}$ in this case correspond to elements of $\boldsymbol{<1|Op^\dagger S|\alpha\alpha'>}$, again those which are zero are left out. The integer array ***I*** is used to relate the index $\alpha\alpha'$ to the index ***p***. Supplied with the operator $\boldsymbol{\sigma(t_o)}$, the vector $\boldsymbol{A_p}$ is easily built from ***A***, ***I*** and ***Sm1***. The integer array ***J*** is not utilized and remains NULL.

When the value of ***ls*** is non-zero and the matrix ***siginf*** is filled, the acquisition is using a Redfield type treatment. Now the elements of the vector $\boldsymbol{A_p}$ well be re-indexed elements from formula (3-10).

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

The elements of ***A*** are the same as in the other Liouville treatment, they again correspond to elements of $\boldsymbol{<1|Op^\dagger S|\alpha\alpha'>}$. The integer array ***I*** is used to relate the index $\alpha\alpha'$ to the index ***p***. Supplied with the operator $\boldsymbol{\sigma(t_o)}$, the vector $\boldsymbol{A_p}$ is easily built from ***A***, ***I*** and ***Sm1*** and ***siginf***. The integer array ***J*** is not utilized and remains NULL. When the trace is calculated the constant value of ***trinf*** will

always be added in.

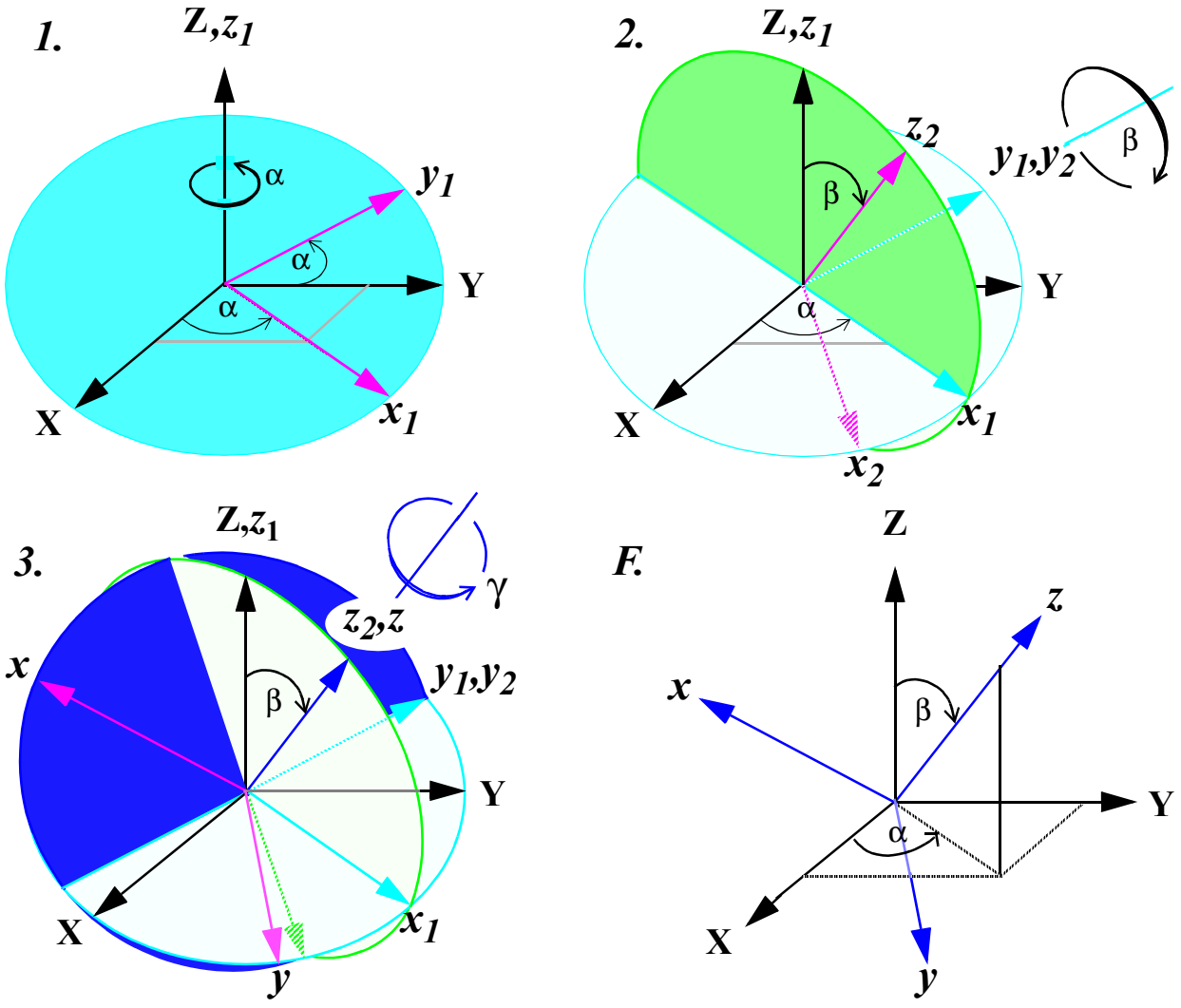
Finally, the internal workings of class acquire is hidden from the average GAMMA user. The private ***class acquire*** function ***create*** is utilized to set up the various array when an ***acquire*** variable is constructed.

4 Euler Angles

4.1 Overview

The class ***EAngles*** embodies a set of three Euler angles, $\{\alpha, \beta, \gamma\}$, which are used to ***define*** an arbitrary rotation. The class structure facilitates the use of such angle sets by providing simple I/O routines, allowing for composite rotations, and consistently maintaining the angles to lie within the ranges $\alpha \in [0, 2\pi)$, $\beta \in [0, \pi]$, and $\gamma \in [0, 2\pi)$. Several modules in GAMMA that perform rotations utilize EAngles as input arguments.

The rotation defined by a set of Euler angles occurs in three steps, each step rotating the coordinate axes in a counter-clockwise fashion about a specific axis. First a rotation of angle α (akin to the spherical angle ϕ) is applied to the existing z-axis. Next a rotation of angle β (akin to spherical angle θ) is applied to the new y-axis. The last rotation is about the new z-axis of angle γ. These 3 steps are shown in the following figure. The two coordinate systems {X,Y,Z} & {x,y,z} are labelled F.



4.2 Available Functions

4.2.1 Document Sections

4.3 Constructors and Assignment

4.3.1 EAngles

Usage:

```
#include <EAngles.h>
EAngles()
EAngles(double alpha, double beta=0, double gamma=0, bool deg=false)
EAngles(const coord& EAs, bool deg=true);
EAngles(const EAngles& EAs)
```

Description:

The *EAngles* constructors are used to create a set of Euler angles.

1. EAngles() - Creates a set of Euler angles, all angles are zero.
2. EAngles(double alpha, double beta=0, double gamma=0, bool deg=false) - New set of Euler angles. The values of alpha, beta, and gamma are assumed to be in ***radians*** unless the flag deg is set true.
3. EAngles(const coord& EAs, bool deg=true) - New set of Euler angles. Here alpha=EAs.x, beta=EAs.y, and gamma=EAs.z. The values in the coordinate are assumed to be in ***degrees*** unless the flag deg is set false.
4. EAngles(const EAngles& EAs) - Constructs a new set of Euler angles, the angles equal to those in EAs.

Note that, regardless of any specified angles, the range of Euler angles is alpha=gamma=[0,2π) and beta=[0.π].

Return Value: EAngles

Examples:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs;                          // A set of 0 Euler angles
  coord XYZ(90, 45.0, -90);             // Coordinate of angles
  EAngles EAxyz(XYZ);                   // A set of Euler angles matching XYZ
  EAngles EAs1(EAxyz);                  // Duplicate of previous set of Euler angles
  EAngles EAs3(PI, .75*PI, 1.154);      // Another set of Euler angles
  }
```

See Also: =

4.3.2 =

Usage:

```
#include <EAngles.h>
EAngles& EAngles operator = (const EAngles& EAs)
```

Description:

The unary operator = (the assignment operator) allows users to easily copy the Euler angle set.

Return Value: EAngles

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs1(PI, .75*PI, 1.154);        // A set of Euler angles
  EAngles EAs3 = EAs1;                    // A duplicate set of Euler angles
  }
```

See Also: EAngles

4.4 Access Functions

4.4.1 alpha

4.4.2 beta

4.4.3 gamma

Usage:

```
#include <Level2/EAngles.h>
double EAngles::alpha()
double EAngles::beta()
double EAngles::gamma()
```

Description:

Functions ***alpha***, ***beta***, and ***gamma*** are used to either get or set the associated Euler angle. The values are **always** set and returned in ***radians***. Note that angles ***alpha*** and ***gamma*** will be restricted to lie within ***[0, 2π)*** and the angle beta to be in the range ***[0, π]***.

Return Value: void or double

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EA;                          // Default Euler angles
  cout << "\n\tAngle Alpha Is "        // Output the angle alpha of EA
       << EA.alpha();
  EA.beta(PI/2.0);                     // Set EA angle beta to 90 degrees
  }
```

See Also: none

4.5 Composite Rotation Functions

4.5.1 *

4.5.2 *=

4.5.3 &=

Usage:

```
#include <Level2/EAngles.h>
EAngles EAngles::operator* (const EAngles& EAs) const;
void EAngles::operator*= (const EAngles& EAs)
void EAngles::operator&= (const EAngles& EAs)
EAngles EAngles::composite(const EAngles& EAs) const;
```

Description:

The operators *, *=, &=, are used to generate a composite set of Euler angles. The result is a set of Euler angles representing the rotation produced if successive Euler rotations had been applied.

Return Value: double or complex

Example:

```
#include <gamma.h>
int main ()
  {
  {
  EAngles EAs1(PI, 0.75*PI, 1.154);      // A set of Euler angles
  EAngles EAs2(0, 0.5*PI, PI);           // A second set of Euler angles
  EAngles EAs = EAs1*EAs2;               // EAngles for rotation EAs2 followed by EAs1
  EAs *= EAs2;                           // EAngles for rotation EAs2 then EAs1 then EAs2
  EAs &= EAs1;                           // EAngles for rotation EAs1 then EAs2 then EAs1 then EAs2
  }
```

4.6 Parameters & Parameter Sets

4.6.1 param

Usage:

```
#include <Level2/EAngles.h>
SinglePar EAngles::param(bool rad=false) const
SinglePar EAngles::param(const string& pn, bool rad=false) const
SinglePar EAngles::param(const string& pn, const string& ps, bool rad=false) const
```

Description:

The function ***param*** generates a single parameter containing the Euler angle definition. In turn, parameter may be written to an external ASCII file that is readable by any GAMMA Euler angle or placed into a ParameterSet for output of multiple parameters. The angles output will be in degrees unless the parameter ***rad*** is set true. The parameter name will be ***EAngles*** unless the string ***pn*** is specified. The parameter comment will be **Euler Angles (deg)** unless the value of ***ps*** is specified.

Return Value: SinglePar

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs2(0, 0.5*PI, PI);          // A second set of Euler angles
  SinglePar par = EAs.param();          // Parameter EAngles (3) : (0,90.0,180.0) - Euler Angles (deg)
  par = EAs.param("xx");                // Parameter xx (3) : (0,90.0,180.0) - Euler Angles (deg)
  par = EAS.param("yy", "yuk");         // Parameter yy (3) : (0,90.0,180.0) - yuk
  }
```

See Also: ParameterSet

4.6.2 ParameterSet

4.6.3 +=

4.6.4 PSetAdd

Usage:

```
#include <Level2/EAngles.h>
EAngles::operator ParameterSet( ) const
void operator+= (ParameterSet& pset, const EAngles& EA)
bool EAngles::PSetAdd(ParameterSet& pset, int idx=-1, bool deg=true, int pfx=-1) const
```

Description:

The operators ***ParameterSet*** and **+=** and the function ***PSetAdd*** are used to place the set of Euler angles into a GAMMA parameter set. The operator ParameterSet() will produce a parameter set containing only one parameter, the set of Euler angles. The opertor += will add the Euler angles, ***EA***, as a single parameter to an existing parameter set, ***pset***. The function ***PSetAdd*** will add the Euler angles to the input parameter set, ***pset***. In the latter case if the value of ***idx*** is set other than ***-1*** then the Euler angles parameter name will have ***(#)*** appended to it where ***#=idx***. If the value of ***pfx*** is set other than ***-1*** then the Euler angles parameter name will have ***[#]*** prepended to it where ***#=pfx***.

Return Value: SinglePar

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs(0, 0.5*PI, PI);                 // A set of Euler angles
  ParameterSet pset = ParameterSet(EAs);      // Parameter set containing the Euler angles
  pset.read("filein.pset");                   // Fill parameter set from parameters in file filein.pset
  pset += EAs;                                // Add set of Euler angles to pset
  EAs.PSetAdd(pset,3,2)                       // Add Euler angles with name [2]EAngles(3) to pset
  }
```

See Also: SinglePar

4.7 Input Output

4.7.1 write

Usage:

#include <Level2/EAngles.h>

void EAngles::write(const string& filename, int idx=-1, bool deg=true) const

Description:

The function ***write*** will create an external ASCII file, filename, that contains the Euler angle set written in GAMMA parameter format. The file may be subsequently read to define the equivalent Euler angles. The parameter will appear in coordinate form as: ***EAngles (3) : { alpha, beta, gamma } - Euler angles (degrees)***, where the angles are in degrees unless the flag ***deg*** is set false. If the value of ***idx*** is set other than -1 the string "(idx)" is appended to the parameter name where idx is an integer value. Euler angles present in an ASCII file in this format may be read into any GAMMA program with the function ***read***.

Return Value: SinglePar

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs(0, 0.5*PI, PI);          // A set of Euler angles
  EAs.write("singleEA.asc");           // Write to ASCII file as a GAMMA parameter
  }
```

See Also: read

4.7.2 read

Usage:

#include <Level2/EAngles.h>

bool EAngles::read(const string& filename, int idx=-1, int warn=2) const

bool EAngles::read(const ParameterSet& pset, int idx=-1, int warn=2) const

Description:

The function read will attempt to set the Euler angles by either reading an external ASCII file or from values found in a GAMMA parameter set. If the file name, filename, is set then the Euler angle set will be read from the file in GAMMA parameter format. The default format is: EAngles (3) : { alpha, beta, gamma } - Euler angles (degrees), where the angles are in degrees unless the flag deg is set false. If the value of idx is set other than -1 the string "(idx)" is appended to the parameter name where idx is an integer value.

Return Value: bool

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs(0, 0.5*PI, PI);          // A set of Euler angles
  EAs.write("singleEA.asc");           // Write to ASCII file as a GAMMA parameter
  }
```

See Also: read

4.7.3 print

4.7.4 <<

Usage:

```
#include <Level2/EAngles.h>
ostream& EAngles::print(ostream& ostr)
ostream& operator << (ostream& ostr, EAngles& EAs)
```

Description:

The function ***print*** and the operator **<<** will output the Euler angles ***EAs*** in a formatted manner to the specified output stream ***ostr***.

Return Value: Output Stream

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs;                    // A set of Euler angles (0,0,0)
  EAs.read("filein.pset");        // Read in angles from filein.pset
  EAs.print(cout);                // Write Euler angles to output stream cout
  cout << EAs;                    // Write Euler angles to standard output (also cout)
  }
```

See Also: read, write

4.8 Auxiliary Functions

4.8.1 equal

Usage:

#include <Level2/EAngles.h>

bool EAngles::equal(const EAngles& EAs, const CUTOFF=1.e-10) const

Description:

The function ***equal*** determines whether two set of Euler angles will produce the same rotation. This is not done by a comparison of angles but by a comparison of the rotation produced. The value of CUTOFF specifies how close of tolerance is used in determining equality.

Return Value: bool

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs;
  EAs.read("filein.pset");
  EAs.print(cout);
  cout << EAs;
  }
```

See Also: none

4.8.2 inverse

Usage:

#include <Level2/EAngles.h>

EAngles EAngles::inverse() const

Description:

The function ***inverse*** returns a set of Euler angles that describes the rotation that is the inverse of the current Euler angle set. Often the inverse of {a,b,g} is simply {-g,-b,-a}, but since there is a restriction on range of values any individual angle may span, an alternative set may be returned.

Return Value: bool

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs(0, 0.5*PI, PI);          // A set of Euler angles
  EAngles EAsInv = EAs.inverse();      // The inverse set of Euler angles
  }
```

See Also: none

4.8.3 RMx

Usage:

#include <Level2/EAngles.h>
matrix EAngles::RMx() const

Description:

The function ***Rmx*** returns a 3x3 matrix representing the rotation defined by the Euler angles.

Return Value: matrix

Example:

```
#include <gamma.h>
int main ()
  {
  EAngles EAs(0, 0.5*PI, PI);          // A set of Euler angles
  matrix R = EAs.RMx();                // 3x3 Rotation matrix
  }
```

See Also: none

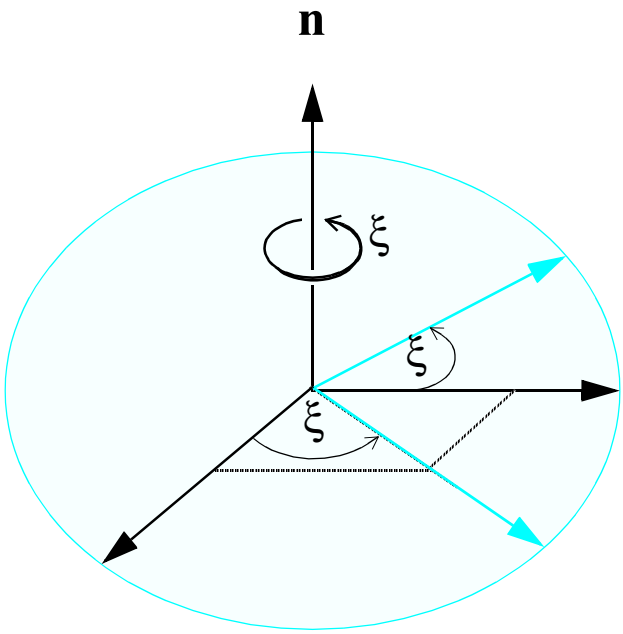
4.9 Description

4.9.1 Introduction

In many areas of physics, Euler angles are used to define an arbitrary rotation. In GAMMA, class ***EAngles*** does exactly the same, i.e. it ***defines an arbitrary rotation***. The class does NOT perform rotations, it is used by other GAMMA classes and functions that perform rotations.

Class ***EAngles*** is designed to facilitate the storage and utilization of three Euler angles $\{\alpha, \beta, \gamma\}$. Herein, we simply provide users with the ability to readily manipulation Euler angles: declare, compare, alter, and view them. They may also be written to and read from external ASCII files. One of the nicer features of ***EAngles*** is the ability to form Euler angles for composite rotations through the use of quaternions (see class quatern).

Each Euler angle specifies a rotation about a particular axis. These will be counter-clockwise rotation rotations that follow the "right-hand-rule": when one places the palm of the right hand on the axis with the thumb in the direction of the positive axis, the rotation occurs in the direction of the fingers as they are wrapped around the axis. For example, the figure below depicts a rotation of angle ξ about the axis ***n***.



Note that the rotation takes the original ***axes*** into new axes, a so called ***passive*** rotation. That is, as viewed by the reader, any points would appear to remain static in space while the rotation occurs. Alternatively, when viewed from an axis in the plane perpendicular to the rotation any points would appear to move in the opposite direction of the rotation. After the rotation is complete any points will then be expressed relative to the new axes. Hence, a point a (1,0,0) will become (0,-1,0) after a 90 degree rotation about the z-axis because that is it's position relative to the rotated axes. Similarly, this same rotation will convert point (0,1,0) into (1,0,0) while point (0,0,1) will remain unchanged.

0.0.3 Rotations Using Euler Angles

Many problems in magnetic resonance reference more than one coordinate system. For example, use of laboratory frame axes, tensor principle axes, molecular fixed axes, and diffusion axes are all quite common. It is thus a fundamental necessity to detail the transformation between coordinate systems, a non-trivial problem. Consider the transformation from a coordinate system {X,Y,Z} into a new coordinate system {x,y,z} as shown in the following figure.

Two Sets of Axes Related Via A Coordinate System Transformation

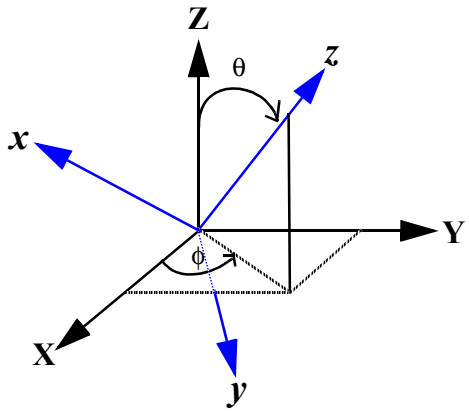


Figure 2-1 Two sets of right handed Cartesian axes, {X,Y,Z} and {x,y,z}. The spherical angles depict the orientation of the z-axis with respect to the {X,Y,Z} system. Note that in the figure only the z-axis orientation if specified by the two angles. A third angle is needed to characterize the orientation of the x and y axes.

In general, the transformation of one coordinate system into another which shares the same origin can be accomplished by the application of three[1] successive rotations. The set of three rotation angles that coincide with these three rotations are called ***Euler*** angles. Typical nomenclature used to designate these angles are $\{\alpha, \beta, \gamma\}$, or $\{\theta, \phi, \chi\}$, or even just Ω for all three. The $\{\theta, \phi, \chi\}$ nomenclature can be advantageous nice because the angles θ and ϕ relate to the standard spherical angles.

Rotations involving Euler angles are defined in the same manner as other authors[2]. The rotations are applied in specific order and occur about specific (and different) axes. They are as follows:

- Counter-clockwise rotation about the Z axis[3] by angle ϕ (or α).
- Counter-clockwise rotation about the new y1-axis by angle θ (or β).
- Counter-clockwise rotation about the new z-axis[4] by angle χ (or γ).

All GAMMA functions using objects of type EAngles must adhere to this definition. Each rota-

1. If the proper rotation axis could be found, only one rotation would suffice. For two arbitrarily oriented sets of axes this is untenable. Often only two rotations suffice, but that lacks the generality required here.
2. In particular, we coincide with two standard texts on spin angular momentum: Brink &Satchler and Zare.
3. According to Zare, this rotation takes the original Y axis to the “line of nodes”, so the next rotation is about the “line of nodes”.
4. The last rotation takes the “line of nodes” into the final y-axis.

tion produces a new set of axes and the subsequent rotation axis belongs to the most recent axis set. Using the angles θ, ϕ, and χ, the following figure explicitly shows how to relate the two coordinate systems shown in the previous figure.

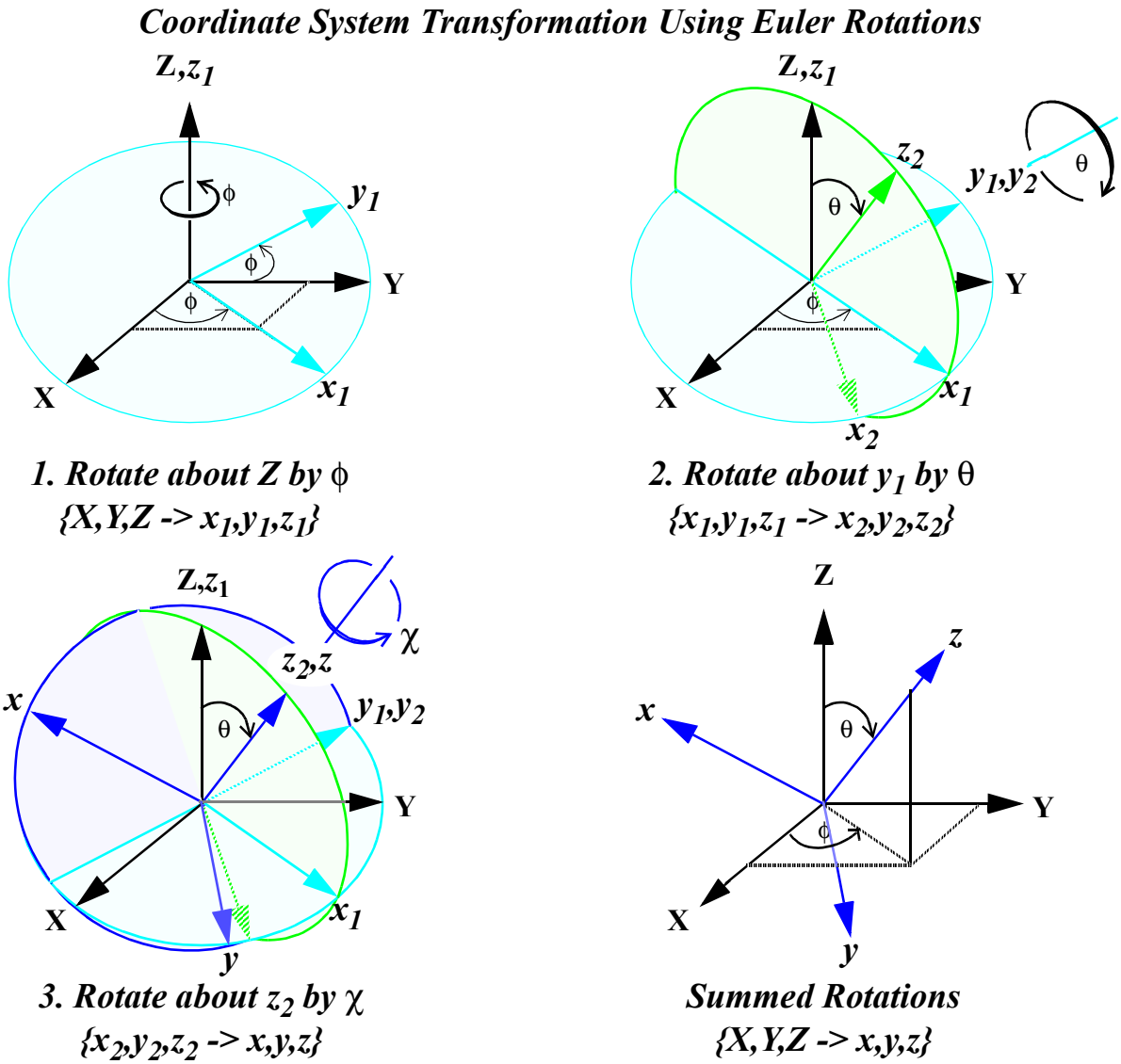


Figure 2-1 After the first rotation the x_1-axis is the projection of the desired z-axis into the original XY-plane. After the 2nd rotation, the z_2-axis in equivalent to the desired z-axis. Furthermore, the x_2 and y_2 axes reside in the plane of the desired x & y axes. The third & final rotation takes x_2 into x and y_2 into y. Angles θ & ϕ are polar angles of the desired z axis relative to the original {X,Y,Z} axes.

To move from one set of axes to another, first get the polar angles of the new z-axis with respect to the original axes, perform the two rotations with those angles, the rotate about the z-axis (by χ) until the x and y axis are at the desired location.

4.10 Rotation Matrices

Euler rotations can be set into 3x3 matrices that act on Cartesian coordinates. Explicit matrices for each of the three rotations are

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \overset{R_Z(\alpha)}{\begin{bmatrix} \cos\alpha & \sin\alpha & 0 \\ -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \qquad \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \overset{R_N(\beta)}{\begin{bmatrix} \cos\beta & 0 & -\sin\beta \\ 0 & 0 & 0 \\ \sin\beta & 0 & \cos\beta \end{bmatrix}} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} \qquad \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \overset{R_z(\gamma)}{\begin{bmatrix} \cos\gamma & \sin\gamma & 0 \\ -\sin\gamma & \cos\gamma & 0 \\ 0 & 0 & 1 \end{bmatrix}} \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} \tag{4-1}$$

By successively applying the three Euler rotations we obtain

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = R_z(\gamma) R_N(\beta) R_Z(\alpha) \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} \tag{4-2}$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\beta\cos\gamma - \sin\alpha\sin\gamma & \sin\alpha\cos\beta\cos\gamma + \cos\alpha\sin\gamma & -\sin\beta\cos\gamma \\ -\cos\alpha\cos\beta\sin\gamma - \sin\alpha\cos\gamma & -\sin\alpha\cos\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\beta\sin\gamma \\ \cos\alpha\sin\beta & \sin\alpha\sin\beta & \cos\beta \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

Keep in mind that these are active rotations wherein the axes are rotating by the specified angles. The coordinates appear to rotate in the opposite direction.

The inverse rotation matrix is generated by reversing the signs of the angles and applying the rotations in the opposite order.

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = R_Z(-\alpha) R_N(-\beta) R_z(-\gamma) \begin{bmatrix} x \\ y \\ z \end{bmatrix} \tag{4-3}$$

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \begin{bmatrix} \cos\alpha\cos\beta\cos\gamma - \sin\alpha\sin\gamma & -\cos\alpha\cos\beta\sin\gamma - \sin\alpha\cos\gamma & \cos\alpha\sin\beta \\ \sin\alpha\cos\beta\cos\gamma + \cos\alpha\sin\gamma & -\sin\alpha\cos\beta\sin\gamma + \cos\alpha\cos\gamma & \sin\alpha\sin\beta \\ -\sin\beta\cos\gamma & \sin\beta\sin\gamma & \cos\beta \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

0.0.4 Angular Momentum

A rotation of angle ϕ about an axis $\hat{v}$ is related to angular momentum $\vec{J}$ by

$$R_{\hat{v}}(\phi) = e^{-i\phi \vec{J} \bullet \hat{v}} \tag{0-1}$$

where ϕ is the angle of rotation. The three Euler rotations (depicted in the last figure) are then given by

$$R(\varphi, \theta, \chi) = R_z(\chi) R_{y_1}(\theta) R_Z(\varphi) = e^{-i\chi J_z} e^{-i\theta J_{y_1}} e^{-i\varphi J_Z} \tag{0-2}$$

It is quite difficult to work with this equation since one usually does not have access to operators such as $\boldsymbol{J_z}$ and $\boldsymbol{J_y}$ because they are only indirectly related the initial and final axis systems. By a simple substitution, we can obtain a similar formula but one which relates all of the rotations to the ***original*** axis system.

$$R(\varphi, \theta, \chi) = R_Z(\varphi) R_Y(\theta) R_Z(\chi) = e^{-i\varphi J_Z} e^{-i\theta J_Y} e^{-i\chi J_Z} \tag{0-3}$$

Note the reversal in rotation application order as well as the axes about which rotations occur. This suggests an alternative picture of how Euler rotations occur.

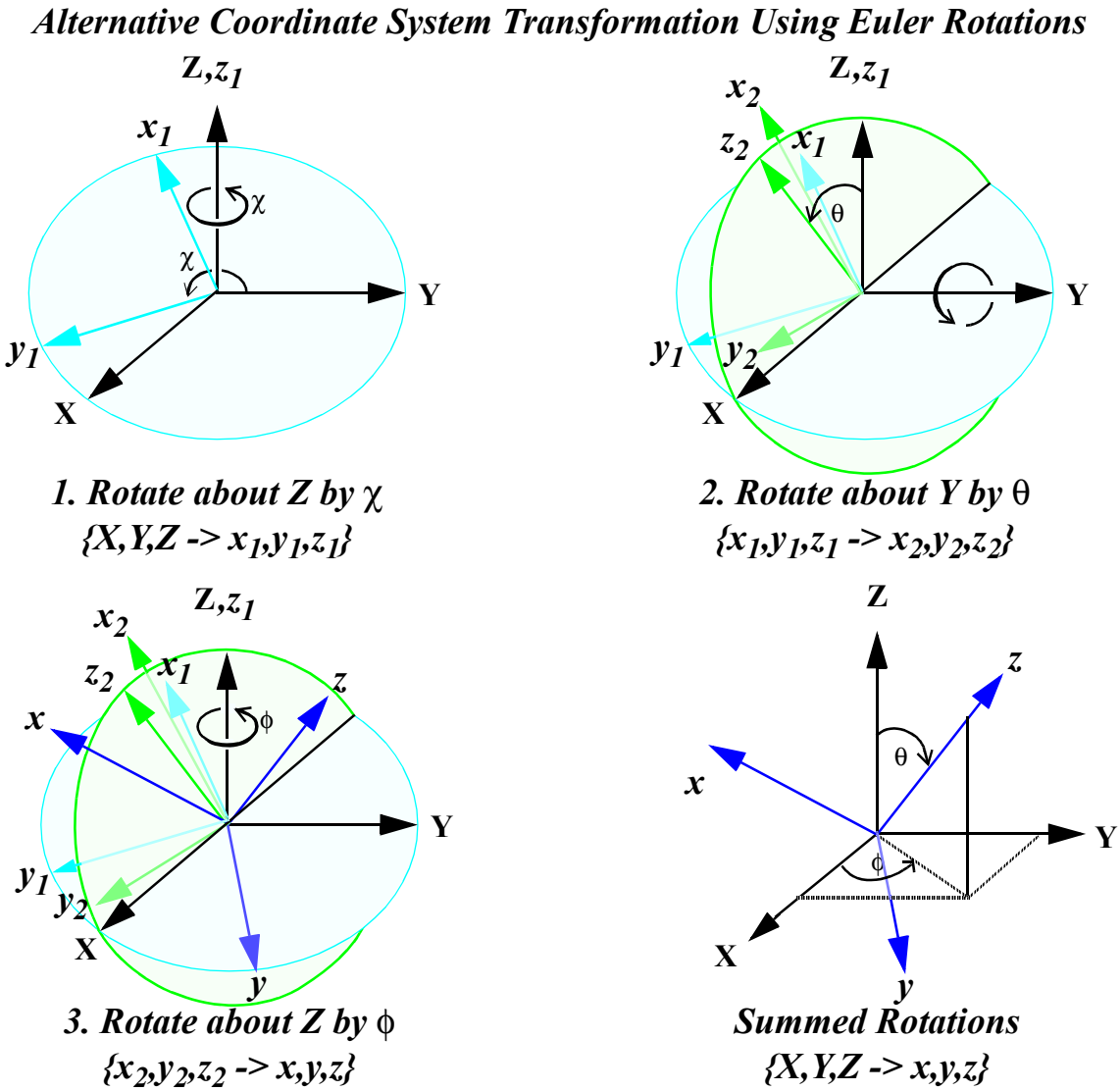


Figure 2-1 The first rotation occurs about the Z-axis by angle χ. It is not obvious how this angle relates the initial and final coordinate systems. The second rotation places the desired z-axis (at this point the z_2-axis) so that it has the polar angle θ relative to the starting coordinate system. The third and final rotation insures that the desired z-axis has the polar angle ϕ relative to the starting coordinate system. Note that the angles θ and ϕ are the polar angles of the desired z axis with respect to the original {X,Y,Z} axes.

0.0.5 References

[1] Brink, D.M. and Satchler, G.R. (1962), Angular Momentum, Clarendon Press, Oxford.

[2] Zare, R.N. (1988), Angular Momentum, John Wiley & Sons, New York.

5 Class Quatern

5.1 Overview

The class quatern is a data type for a quaternion. Quaternions are similar to a Euler angles (see class EAngles) in that each quaternion describes a rotation in three dimensional space. Unlike a Euler angles, quaternions may be readily used to produce composite rotations. That is, if two quaternions represent two successive rotations a third quaternion representing the composite rotation can easily be obtained. Two uses of Quaternions in magnetic resonance are the generation of composite pulses and the manipulation of oriented Hamiltonians in multiple reference frames.

5.2 Available Quaternion Functions

write	- Write to ASCII file	page 91
read	- Read from ASCII file	page 92
print	- Print to output stream	page 91
<<	- Print to output stream	page 91

5.3 Algebraic Operators

5.3.1 quatern

Usage:

```
#include <Quaternion.h>
quatern::quatern();
quatern::quatern(const coord&   ABG, bool inv=false);
quatern::quatern(const EAngles& EA, bool inv=false);
quatern::quatern(const quatern& Qrt, bool inv=false);
quatern::quatern(const ParameterSet& pset, int idx=-1, int warn=2);
quatern::quatern(double QA, double QB, double QC, double QD, bool inv=false);
```

Description:

The function ***quatern*** is used to create a quaternion. The quaternion can be constructed as a default quaternion, a quaternion for a normal rotation or for an inverse rotation. The inverse is achieved by setting the value of boolean flag ***inv*** to true.

1. quatern() - Creates an "empty"quaternion which has the default values Qrt = {0,0,0,1}.
2. quatern(coord &EA, int inv=0) - Called with the set of Euler angles EA (contained in a coordinate point) and an optional inverse flag, inv, the quaternion equivalent to the Euler angle set will be constructed. If the inverse flag is set to non-zero, the quaternion equivalent to the inverse rotation is constructed.
3. quatern(quatern &Qrt) - Called with another quaternion Qrt and an optional inverse flag, inv, this function constructs an identical quaternion to the input quaternion. If the inverse flag is set to non-zero, the quaternion equivalent to the inverse rotation is constructed.
4. quatern(double A, double B, double C, double D, int inv=0) - Called with four double precision numbers and an optional inverse flag, inv, the four doubles are taken to be the components of the returned quaternion. If the inverse flag is set to non-zero, the quaternion inverse is constructed.

Return Value: Constructor

Examples:

```
#include <gamma.h>
int main()
  {
  #include <Quaternion.h>
  quatern Qtr();                    // Construct a NULL quaternion
  quatern Qtr1(0, 1, 0, 0);         // Construct Qtr1 by specifying {A,B,C,D.
  quatern Qtr2(Qtr1);               // Construct Qtr2 identical to Qtr1
  coord EA(0,180.0,0);              // Construct a coordinate containing Euler angles.
  quatern Qtr3(EA);                 // Construct Qtr3 from EA, should be same as Qtr1
  }
```

Mathematical Basis:

Each quaternion consists of four components, herein called ***A***,***B***,***C***, and ***D***, ***Qrt = {A, B, C, D}***. When a quaternion is constructed without arguments, these four values are set to NULL; ***Qrt() -> Qrt = {0,0,0,0}***. When constructed with four values, the four values are set equal to the input; ***Qrt(A, B, C, D) -> Qrt = {A,B,C,D***}. When constructed from another quaternion, the four components are set equal between the two: ***Qrt(Qrt1) -> Qrt = {A1,B1,C1,D1},*** where ***Qrt1 = {A1,B1,C1,D1}***.

See Also: =

5.3.2 =

Usage:

#include <Quaternion.h>
void quatern operator = (quatern& Q)
void quatern operator = (coord& EA)
void quatern operator = (EAngles& EA)

Description:

The unary operator = (the assignment operator) allows for the setting of one quatern to another quatern. If the quatern being assigned to exists it will be overwritten by the assigned quatern. This function also allows one to directly set a coordinate point equal to the quaternion. In this latter case it is assumed that the point contains a set of Euler angles and the quaternion equivalent is computed.

Return Value: void

Examples:

```
#include <gamma.h>
int main()
  {
  #include <Quaternion.h>
  quatern Qtr();                    // Construct a NULL quaternion
  quatern Qtr1(0, 1, 0, 0);         // Construct Qtr1 by specifying {A,B,C,D}.
  Qtr = Qtr1;                       // Set Qtr identical to Qtr1
  coord EA(0,180.0,0);              // Construct a coordinate containing Euler angles.
  Qtr1 = quatern(EA);               // Set Qtr1 now to quaternion equal to EA.
  }
```

See Also: quatern

5.4 Access

5.4.1 A
5.4.2 B
5.4.3 C
5.4.4 D

Usage:

```
#include <Quaternion.h>
double quatern::A()
double quatern::B()
double quatern::C()
double quatern::D()
```

Description:

The functions ***A***, ***B***, ***C***, ***D*** return the quaternion components of the same name.

Return Value: double

Examples:

```
#include <gamma.h>
int main()
  {
  quatern Qtr(0, 1, 0, 0);                  // Construct Qtr by specifying {A,B,C,D}.
  double a = Qtr.A();                       // Get the value of A
  cout << "\n\tB = " << Qtr.B();            // Output the value of B
  }
```

5.5 Euler Angles

5.5.1 alpha
5.5.2 beta
5.5.3 gamma

Usage:

#include <Quaternion.h>

double quatern::alpha(double beta=-1000)

double quatern::beta( )

double quatern::gamma(double beta=-1000, double alpha=-1000)

Description:

The functions ***alpha***, ***beta***, and ***gamma*** return the Euler angles of the same name associated with the input quaternion. Values are returned in degrees. The computation of alpha is more efficient if beta is known (See function beta), but its input is not mandatory. Similarly, the computation of gamma is more efficient if the other two Euler angles, beta and alpha, are known, but their input is not mandatory.

Return Value: double

Examples:

```
#include <gamma.h>
int main()
  {
  quatern Qtr(0, 1, 0, 0);                    // Construct Qtr by specifying {A,B,C,D}.
  cout << "\nThe angle beta is " << Qtr.beta();// Output Euler angle beta for this quaternion.
  quatern Qtr();                              // Construct a NULL quaternion, Qtr.
  coord EA(10.0,180.0,45.0);                  // Construct a coordinate containing Euler angles.
  Qtr = quatern(EA);                          // Set Qtr now to quaternion equal to EA.
  cout << "\nAlpha is " << Qtr.alpha();       // Output Euler angle alpha, hopefully 10 degrees.
  Qtr = quatern(EA);                          // Set Qtr now to quaternion equal to EA.
  cout << "\nGamma is " << Qtr.gamma();      // Output Euler angle alpha, hopefully 45 degrees.
  quatern Qtr();                              // Construct a NULL quaternion, Qtr.
  Qtr = quatern(EA);                          // Set Qtr now to quaternion equal to EA.
  cout << "\nGamma is " << Qtr.gamma();      // Output Euler angle alpha, hopefully 45 degrees.
  }
```

See Also: EA

5.5.4 EA

Usage:

#include <Quaternion.h>

EAngles quatern::EA()

Description:

The function ***EA*** returns a set of Euler angles that produce the equivalent rotation as the quaternion.

Return Value: EAngles

Example:

```
#include <gamma.h>
int main()
  {
  #include <Quaternion.h>
  quatern Qtr();                          // Construct a NULL quaternion, Qtr.
  coord EA(10.0,180.0,45.0);              // Construct a coordinate containing Euler angles.
  Qtr = quatern(EA);                      // Set Qtr now to quaternion equal to EA.
  cout << "\nAngles are" << Qtr.EA();     // Output Euler angles, hopefully 10, 180, & 45.
  }
```

See Also: alpha, beta, gamma

5.5.5 ABG

Usage:

```
#include <Quaternion.h>
coord quatern::ABG()
```

Description:

The function ***ABG*** returns a coordinate containing the respective Euler angles that produce the equivalent rotation as the quaternion. The elements of the coordinate will be in degrees.

Return Value: EAngles

Example:

```
#include <gamma.h>
int main()
  {
  #include <Quaternion.h>
  quatern Qtr();                          // Construct a NULL quaternion, Qtr.
  coord EA(10.0,180.0,45.0);              // Construct a coordinate containing Euler angles.
  Qtr = quatern(EA);                      // Set Qtr now to quaternion equal to EA.
  cout << "\nAngles are" << Qtr.EA();     // Output Euler angles, hopefully 10, 180, & 45.
  }
```

See Also: alpha, beta, gamma

5.6 Rotations

5.6.1 *

5.6.2 *=

5.6.3 &=

Usage:

```
#include <Quaternion.h>
quatern quatern::operator* (const quatern& Q) const;
void quatern::operator*= (const quatern& Q);
void quatern::operator&= (const quatern& Q);
```

Description:

The quaternion operators ***, ***=**, and ***&=*** are used to generate quaternions for composite rotations. The given the input quaternion, ***Q***, will be represent the first rotation in functions ***** and ***=**, but will be the second rotation in the function ***&=***.

Return Value: void or quaternion

Examples:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // Quaternion for rotation
  quatern Q2(72.9, 345.1,16.0);         // Quaternion for rotation
  quartern Q2Q1 = Q2 * Q1;              // Quaternion for rotation 1 followed by rotation 2
  quartern Q1 *= Q2;                    // Q1 now for rotation 2 followed by rotation 1
  quartern Q1 & =  Q2;                  // Q1 now for rotation 1 followed by rotation 2
  }
```

See Also: RMx

5.6.4 composite

Usage:

```
#include <Quaternion.h>
quatern composite(coord &EA1, coord &EA2);
quatern composite(quatern &Qrt1, quatern &Qrt2);
```

Description:

The function ***composite*** returns a quaternion which represents the composite rotation resulting from two successive rotations. The individual rotations may be input as either two Euler angle sets or a two quaternions.

Return Value: quaternion

Example(s):

```
#include <gamma.h>
int main()
  {
```

```
}
```

See Also: Rotmx

5.6.5 RMx

Usage:

```
#include <Quaternion.h>
matrix quatern::RMx();
```

Description:

The function ***RMx*** returns a 4x4 rotation matrix for a Quaternion in accordance with equation (5-4). The product of this matrix times an initial quaternion produces a new quaternion which is the composite of the two rotations.

Return Value: matrix

Example:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // Quaternion for rotation
  matrix R = Q1.RMx();                  // Rotation matrix for quaternion (4x4)
  quatern Q2(72.9, 345.1,16.0);         // Quaternion for rotation
  quartern Q2Q1 = Q2 * Q1;              // Quaternion for rotation 1 followed by rotation 2
  quatern Q1Q2 = R*Q1;                  // Quaternion for rotation 2 followed by rotation 1
  }
```

See Also: composite

5.7 General Functions

5.7.1 norm

Usage:

#include <Quaternion.h>
double quatern::norm() const

Description:

The function ***norm*** returns the sum of the squares over the four quaternion elements. In principle this should always return the value of 1 in accordance with equation (5-2).

Return Value: double

Example:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // Quaternion for rotation
  double nval = Q1.norm();              // Norm of the quaternion
  cout << "\n\tDifference From Norm 1: "
       << 1.0-nval << "\n\n";
  }
```

See Also:

5.7.2 inverse

Usage:

#include <Quaternion.h>
double quatern::inverse() const

Description:

The function ***inverse*** returns the quaternion representing the inverse rotation. The inverse is obtained by simply negating the value of D in the input quaternion.

Return Value: quaternion

Example:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // A quaternion
  quatern Q1inv = Q1.inverse();         // The inverse of the 1st quaternion
  quatern Q = Q1inv*Q1;                 // This should be {0, 0, 0, 1}, no rotation
  }
```

See Also:

5.8 Parameters & Parameter Sets

5.8.1 param

Usage:

#include <Quaternion.h>

SinglePar quatern::param() const

SinglePar quatern::param(const string& pn) const

SinglePar quatern::param(const string& pn, const string& ps) const

Description:

The function ***param*** generates a single parameter containing the quaternion definition. In turn, the parameter may be written to an external ASCII file that is readable by any GAMMA quaternion or placed into a ParameterSet for output of multiple parameters. The parameter name will be ***Quaternion*** unless the string ***pn*** is specified. The parameter comment will be ***A Quaternion*** unless the value of ***ps*** is specified.

Return Value: SinglePar

Example:

```
#include <gamma.h>
int main ()
  {
  quatern Q(90.0, 45.0, 1.0);          // Quaternion for rotation
  SinglePar par = Q.param();           // Parameter Quaternion (3) : (A,B,C) - A Quaternion
  par = Q.param("xx");                 // Parameter xx (3) : (A,B,C) - A Quaternion
  par = Q.param("yy", "yuk");          // Parameter yy (3) : (A,B,C) - yuk
  }
```

See Also: ParameterSet

5.8.2 ParameterSet

5.8.3 +=

5.8.4 PSetAdd

Usage:

#include <Quaternion.h>

quatern::operator ParameterSet( ) const

void operator+= (ParameterSet& pset, const quatern& Q)

bool quatern::PSetAdd(ParameterSet& pset, int idx=-1, int pfx=-1) const

Description:

The operators ***ParameterSet*** and += and the function ***PSetAdd*** are used to place a quaternion into a GAMMA parameter set. The operator ParameterSet() will produce a parameter set containing only one parameter, the quaternion. The opertor += will add the quaternion, ***Q***, as a single parameter to an existing parameter set, ***pset***. The function ***PSetAdd*** will add the quaternion to the input parameter set, ***pset***. In the latter case if the value of ***idx*** is set other than ***-1*** then the quaternion parameter name will have ***(#)*** appended to it where ***#=idx***. If the value of ***pfx*** is set other than ***-1*** then the quaternion parameter name will have ***[#]*** prepended to it where ***#=pfx***.

Return Value: SinglePar

Example:

```
#include <gamma.h>
int main ()
  {
  quatern Q(90.0, 45.0, 1.0);                 // Quaternion for rotation
  ParameterSet pset = ParameterSet(Q);        // Parameter set containing the quaternion
  pset.read("filein.pset");                   // Fill parameter set from parameters in file filein.pset
  pset += Q;                                  // Add quaternion to pset
  Q.PSetAdd(pset,3,2)                         // Add quaternion with name [2]Quaternion(3) to pset
  }
```

See Also: SinglePar

5.9 Input/Output

5.9.1 print

5.9.2 <<

Usage:

```
#include <Quaternion.h>
std::ostream& quatern::print(std::ostream& ostr, bool nf=true, bool hdr=true) const;
friend std::ostream& operator << (std::ostream& ostr, const quatern& Q);
```

Description:

The function ***print*** and the operator **<<** will send information regarding the quaternion ***Q*** to the specified output stream ***ostr***. When using the ***print*** function the boolean ***hdr*** set ***false*** will suppress any header information. If the flag ***nf*** is set ***false*** the quaternion norm will not be output.

Return Value: none

Examples:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // A quaternion
  Q.print(cout, false,false);           // Output quaternion to standard output, no header no norm
  cout << Q;                            // Output quaternion to standard output, with header & norm
  }
```

See Also: write

5.9.3 write

Usage:

```
#include <Quaternion.h>
bool quatern::write(const std::string& fileout, int idx=-1,int pfx=-1,int warn=2) const;
bool quatern::write(    std::ofstream& ofstr, int idx=-1,int pfx=-1,int warn=2) const;
```

Description:

The function ***write*** will write the quaternion to an output ASCII file with name ***fileout*** or output file stream ***ofstr***. The output format of the quaternion will be that of a GAMMA single parameter and match that used by the quaternion read function. If the value of ***idx*** is set other than the default of ***-1*** then ***(#)*** will be added to the output quaternion parameter name where ***#=idx***. The flag warn determines what should be done if the output fails: ***0 = do nothing***, ***1 = issue warnings***, ***other = stop program execution***. The function will return ***true*** if the write was successful.

Return Value: bool

Examples:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);          // A quaternion
```

```
string newfile("quat.pset");          // Name for new output file
Q.write(newfile);                     // Create new file quat.pset containing Q as a GAMMA parameter
ofstream ofstr(newfile.c_str());      // Open an output file stream attached to quat.pset
Q.write(ofstr, 1);                    // Write Q into output file stream with index of 1
}
```

See Also: read

5.9.4 read

Usage:

```
#include <Quaternion.h>
bool quatern::read(const std::string& filein, int indx=-1, int warn=2)
bool quatern::read(const ParameterSet& pset,   int indx=-1, int warn=2)
```

Description:

Functon ***read*** is used to read in a single quaternion from either an external ASCII file ***filein*** or a GAMMA parameter set ***pset***. The function returns ***true*** if the quaternion has been successfully read. The value of index will affect which parameter name is sought, the suffix [indx] is added to the parameter name when indx is not the default value of -1. The value of integer warn sets what the function response should be if the quaternion is not read: 0=do nothing, 1=issue non-fatal warnings, else=exit program after issuing warnings.

Return Value: bool

Exampled:

```
#include <gamma.h>
int main()
  {
  quatern Q1(90.0, 45.0, 1.0);        // A quaternion
  Q.print(cout, false,false);         // Output quaternion to standard output, no header no norm
  cout << Q;                          // Output quaternion to standard output, with header & norm
  }
```

See Also: write

5.10 Description

There are several functions supplied with GAMMA which rotate either objects or coordinate systems. Some GAMMA classes even have intrinsic rotation capabilities: two such examples are spin operators and spatial tensors. Typically, arbitrary rotations are defined in terms of Euler angles. This is because it generally takes three rotations to transform one coordinate system into another and there are three Euler angles, one per each rotation. When multiple rotations are performed with Euler angles it is difficult to keep track of the net rotation. That is, it is difficult to ascertain a set of three Euler for a composite rotation. Such is not the case for Quaternions. Quaternions are another formulation of rotations, a formulation that has no difficulty in determining composite rotations.

5.10.1 Components From Euler Angles

The quaternion functions provided herein are based on an article by Blümich and Spiess[1]. Each quaternion consists of four real numbers, A,B,C and D. These are related to a set of Euler angles alpha, beta and gamma as given by the following formulae (equivalent to equation labeled [7] in the reference).

$$\begin{aligned} A &= -\sin\frac{\beta}{2}\sin\frac{\alpha-\gamma}{2} \qquad B = \sin\frac{\beta}{2}\cos\frac{\alpha-\gamma}{2} \\ C &= \cos\frac{\beta}{2}\sin\frac{\alpha+\gamma}{2} \qquad D = \cos\frac{\beta}{2}\cos\frac{\alpha+\gamma}{2} \end{aligned} \tag{5-1}$$

Although there are four terms to each quaternion, there are only three unique parameters represented because of the constraint

$$A^2 + B^2 + C^2 + D^2 = 1. \tag{5-2}$$

It will be worthwhile to know when the quaternion components are zero. From the previous definitions,

$$\begin{aligned} A(\alpha,\beta,\gamma): &\quad 0 = A(\alpha, 2n\pi, \gamma) = A(\alpha, \beta, \alpha \pm 2n\pi) \\ B(\alpha,\beta,\gamma): &\quad 0 = B(\alpha, 2n\pi, \gamma) = B(\alpha, \beta, \alpha \pm [2n+1]\pi) \\ C(\alpha,\beta,\gamma): &\quad 0 = C(\alpha, [2n+1]\pi, \gamma) = C(\alpha, \beta, \alpha \pm 2n\pi) \\ D(\alpha,\beta,\gamma): &\quad 0 = D(\alpha, [2n+1]\pi, \gamma) = D(\alpha, \beta, [2n+1]\pi) \end{aligned} \tag{5-3}$$

where n is a non-negative integer. In GAMMA, Euler angles are restricted to the regions $\alpha \in [0, 2\pi]$, $\beta \in [0, \pi]$, and $\gamma \in [0, 2\pi]$. This places even further limitations on when the quaternion components will be zero as it forces n = 0.

1. JMR, **61**, 356-362 (1985), *Quaternions as a Practical Tool for the Evaluation of Composite Rotations.*

5.10.2 Composite Rotations

Quaternions are useful due to the ease with which one may determine the quaternion for the sum of successive rotations from the quaternions of the individual rotations. For two rotations (labeled 0 and 1), the quaternion for the composite rotation (labeled 2) is given by (this is equation [9] in the citation)

$$Q_2 = R_1 Q_0$$

$$\begin{bmatrix} A_2 \\ B_2 \\ C_2 \\ D_2 \end{bmatrix} = \begin{bmatrix} D_1 & -C_1 & B_1 & A_1 \\ C_1 & D_1 & -A_1 & B_1 \\ -B_1 & A_1 & D_1 & C_1 \\ -A_1 & -B_1 & -C_1 & D_1 \end{bmatrix} \begin{bmatrix} A_0 \\ B_0 \\ C_0 \\ D_0 \end{bmatrix} = \begin{bmatrix} A_0D_1 - B_0C_1 + C_0B_1 + D_0A_1 \\ A_0C_1 + B_0D_1 - C_0A_1 + D_0B_1 \\ -A_0B_1 + B_0A_1 + C_0D_1 + D_0C_1 \\ -A_0A_1 - B_0B_1 - C_0C_1 + D_0D_1 \end{bmatrix} \tag{5-4}$$

where R_1 is a rotation matrix formed from the quaternion Q_1. There are no analogous formulas which produce the Euler angles of summed rotations from the individual rotation Euler angle sets, that is, there is no simple Euler angle equivalent to the previous equation. Since composite rotations are evaluated so readily via (5-4), it would be very useful to have the inverses of equations (5-1) so that one could readily convert back to Euler angles from the values of the quaternion. Unfortunately these inverse formulas are not straightforward. The following is the derivation for the inverse equations desired. We will remind the reader that, in GAMMA, Euler angles are restricted to the regions $\alpha \in [0, 2\pi]$, $\beta \in [0, \pi]$, and $\gamma \in [0, 2\pi]$.

5.10.3 Euler Angle β From Quaternion

We begin by first deriving the Euler angle β in terms of A,B,C, and D. Taking the squares of each quaternion element produces the following equations.

$$A^2 = \sin^2\left(\frac{\beta}{2}\right)\sin^2\left(\frac{\alpha-\gamma}{2}\right) \qquad B^2 = \sin^2\left(\frac{\beta}{2}\right)\cos^2\left(\frac{\alpha-\gamma}{2}\right)$$

$$C^2 = \cos^2\left(\frac{\beta}{2}\right)\sin^2\left(\frac{\alpha+\gamma}{2}\right) \qquad D^2 = \cos^2\left(\frac{\beta}{2}\right)\cos^2\left(\frac{\alpha+\gamma}{2}\right)$$

Combining terms and taking advantage of the identity ***sin²(x) + cos²(x) = 1***,

$$A^2 + B^2 = \sin^2\left(\frac{\beta}{2}\right)\left[\sin^2\left(\frac{\alpha-\gamma}{2}\right) + \cos^2\left(\frac{\alpha-\gamma}{2}\right)\right] = \sin^2\left(\frac{\beta}{2}\right)$$

$$C^2 + D^2 = \cos^2\left(\frac{\beta}{2}\right)\left[\sin^2\left(\frac{\alpha+\gamma}{2}\right) + \cos^2\left(\frac{\alpha+\gamma}{2}\right)\right] = \cos^2\left(\frac{\beta}{2}\right)$$

$$\sqrt{A^2 + B^2} = \sin\left(\frac{\beta}{2}\right) \qquad \sqrt{C^2 + D^2} = \cos\left(\frac{\beta}{2}\right)$$

This results is two equations for determining the angle β from a quaternion's components.

$$2\,\mathrm{asin}\left[\sqrt{A^2+B^2}\right] \;=\; \beta \;=\; 2\,\mathrm{acos}\left[\sqrt{C^2+D^2}\right] \tag{5-5}$$

Since the trigonometric functions are not uniquely valued over the range of possible angles, [-2π, 2π], it can be beneficial to use both formulas in order to clarify which of the possible β's is most appropriate for a given quaternion. Of course, we will restrict our chosen angle to reside within the GAMMA specified Euler angle range, $\beta \in [0, \pi]$, so this multi-value nature is not problematic.

5.10.4 Euler Angles α and γ From Quaternion

Equations for α and γ are algebraically more difficult to obtain and the multivalued nature of these functions must be dealt with here as well. *Assuming that we know the angle* β, we can readily obtain relationships for the sums and differences of α and γ. From the definitions of A-D,

$$\begin{aligned}\alpha-\gamma \;&=\; 2\,\mathrm{asin}\frac{-A}{\sin(\beta/2)} \;=\; 2\,\mathrm{acos}\frac{B}{\sin(\beta/2)}\\ \alpha+\gamma \;&=\; 2\,\mathrm{asin}\frac{C}{\cos(\beta/2)} \;=\; 2\,\mathrm{acos}\frac{D}{\cos(\beta/2)}\end{aligned} \tag{5-6}$$

Of course, care must be employed when the denominators in the inverse trigonometric functions are zero. That is, the above formulae for α-γ contain asin(0/0) at β = 0 and the formulae for α + γ contain (0/0) at β = π. Excluding these two angles (recall $\beta \in [0, \pi]$ in GAMMA), we can generate formulae for α and γ by taking sums and differences of the previous two formulae. There are four possibilities for each angle.

$$\begin{aligned}\alpha \;&=\; \mathrm{asin}\frac{-A}{\sin(\beta/2)}+\mathrm{asin}\frac{C}{\cos(\beta/2)} \;=\; \mathrm{asin}\frac{-A}{\sin(\beta/2)}+\mathrm{acos}\frac{D}{\cos(\beta/2)}\\ &=\; \mathrm{acos}\frac{B}{\sin(\beta/2)}+\mathrm{asin}\frac{C}{\cos(\beta/2)} \;=\; \mathrm{acos}\frac{B}{\sin(\beta/2)}+\mathrm{acos}\frac{D}{\cos(\beta/2)}\\ \gamma \;&=\; -\,\mathrm{asin}\frac{-A}{\sin(\beta/2)}+\mathrm{asin}\frac{C}{\cos(\beta/2)} \;=\; -\,\mathrm{asin}\frac{-A}{\sin(\beta/2)}+\mathrm{acos}\frac{D}{\cos(\beta/2)}\\ &=\; -\,\mathrm{acos}\frac{B}{\sin(\beta/2)}+\mathrm{asin}\frac{C}{\cos(\beta/2)} \;=\; -\,\mathrm{acos}\frac{B}{\sin(\beta/2)}+\mathrm{acos}\frac{D}{\cos(\beta/2)}\end{aligned} \tag{5-7}$$

Since both sine and cosine are multi-valued over the range of α and γ, [0, 360), one needs to use caution in order to determine the best values for a particular quaternion. Take special note that when β = 0 only the sum α + γ can be determined and when β = π only the difference α - γ can be determined. By setting the angle g to zero in these two cases one can use the following.

$$\begin{aligned}\beta = 0\text{:}\qquad \alpha \;&=\; 2\,\mathrm{asin}-A \;=\; 2\,\mathrm{acos}B\\ \beta = \pi\text{:}\qquad \alpha \;&=\; 2\,\mathrm{asin}C \;=\; 2\,\mathrm{acos}D\end{aligned} \tag{5-8}$$

Of course, one can further manipulate these equations to produce more compact formula for α and γ, however it is not clear that will produce any computational efficiency nor additional insight. We begin by taking the products of all the quaternion components.

$$AB = -\sin^2\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right) \qquad AC = -\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right)$$

$$AD = -\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right) \qquad BC = \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right)$$

$$BD = \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right) \qquad CD = \cos^2\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right)$$

There are three trigonometric identities which will be applied at this point. They are

$$\sin x\cos y = \frac{1}{2}[\sin(x+y)+\sin(x-y)]$$

$$\cos x\cos y = \frac{1}{2}[\cos(x+y)+\cos(x-y)],$$

$$\sin x\sin y = \frac{1}{2}[\cos(x-y)-\cos(x+y)]$$

Upon application of these to the previous set of equations, we obtain the following formulae.

$$AB = -\sin^2\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right) = -\frac{1}{2}\sin^2\left(\frac{\beta}{2}\right)\sin(\alpha-\gamma)$$

$$AC = -\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right) = -\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)[\cos(-\gamma)-\cos(\alpha)]$$

$$AD = -\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right) = -\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)[\sin(\alpha)+\sin(-\gamma)]$$

$$BC = \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right) = \frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)[\sin(\alpha)+\sin(\gamma)]$$

$$BD = \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\cos\left(\frac{\alpha-\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right) = \frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)[\cos(\alpha)+\cos(-\gamma)]$$

$$CD = \cos^2\left(\frac{\beta}{2}\right)\sin\left(\frac{\alpha+\gamma}{2}\right)\cos\left(\frac{\alpha+\gamma}{2}\right) = \frac{1}{2}\cos^2\left(\frac{\beta}{2}\right)\sin(\alpha+\gamma)$$

By taking linear combinations of these products the angles α and γ become more isolated. Letting

$$X = \sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right) \qquad A^2+B^2 = \sin^2\left(\frac{\beta}{2}\right) \qquad C^2+D^2 = \cos^2\left(\frac{\beta}{2}\right),$$

the following equations are beginning to appear simplified.

$$-2AB = [A^2 + B^2]\sin(\alpha - \gamma) \qquad 2CD = [C^2 + D^2]\sin(\alpha + \gamma)$$

$$AC + BD = \left[\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\right][-\cos(-\gamma) + \cos(\alpha) + \cos(\alpha) + \cos(-\gamma)] = X\cos(\alpha)$$

$$AC - BD = \left[\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\right][-\cos(-\gamma) + \cos(\alpha) - \cos(\alpha) - \cos(-\gamma)] = -X\cos(-\gamma)$$

$$AD + BC = \left[\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\right][-\sin(\alpha) - \sin(-\gamma) + \sin(\alpha) + \sin(\gamma)] = X\sin(\gamma)$$

$$AD - BC = \left[\frac{1}{2}\sin\left(\frac{\beta}{2}\right)\cos\left(\frac{\beta}{2}\right)\right][-\sin(\alpha) - \sin(-\gamma) - \sin(\alpha) - \sin(\gamma)] = -X\sin(\alpha)$$

An additional combination step must be performed on the first two equations.

$$\text{asin}\left(\frac{-2AB}{A^2 + B^2}\right) = \alpha - \gamma \qquad \text{asin}\left(\frac{2CD}{C^2 + D^2}\right) = \alpha + \gamma$$

This last step is a bit dangerous as it amounts to dividing by zero for some quaternions (*i.e.* at certain angles). Working formulae for obtaining α and γ are now achieved by simple algebra. Forming tangents from the equations which have sines and cosines on one side, the factor X drops out.

$$\frac{AD + BC}{AC - BD} = \frac{X\sin(\gamma)}{-X\cos(-\gamma)} = \frac{\sin(\gamma)}{-\cos(\gamma)} = -\tan(\gamma) \qquad \frac{AD - BC}{AC + BD} = \frac{-X\sin(\alpha)}{X\cos(\alpha)} = -\tan(\alpha)$$

$$\text{atan}\left(\frac{AD + BC}{BD - AC}\right) = \gamma \qquad \text{atan}\left(\frac{BC - AD}{AC + BD}\right) = \alpha \tag{5-9}$$

Taking a linear combination of the other two equations produces a second set of formula for α & γ.

$$\frac{1}{2}\left[\text{asin}\left(\frac{-2AB}{A^2 + B^2}\right) + \text{asin}\left(\frac{2CD}{C^2 + D^2}\right)\right] = \alpha \qquad \frac{1}{2}\left[\text{asin}\left(\frac{2CD}{C^2 + D^2}\right) - \text{asin}\left(\frac{-2AB}{A^2 + B^2}\right)\right] = \gamma \tag{5-10}$$

At this point we have four equations, (5-9) and (5-10), and only two unknowns, α and γ. It appears that the system is over-characterized but this is not the case due to the multivalued nature of the (arc-)trigonometric functions. We must now consider when it is appropriate to apply these equations. Recall that

$$(A^2 + B^2)\big|_{\beta = 0} = \sin^2\left(\frac{\beta}{2}\right)\Big|_{\beta = 0} = 0 \qquad (C^2 + D^2)\big|_{\beta = \pm\pi} = \cos^2\left(\frac{\beta}{2}\right)\Big|_{\beta = \pm\pi} = 0.$$

Furthermore

$$AB|_{\beta = 0} = AC|_{\beta = 0} = AD|_{\beta = 0} = BC|_{\beta = 0} = BD|_{\beta = 0} = 0$$
$$AC|_{\beta = \pm\pi} = AD|_{\beta = \pm\pi} = BC|_{\beta = \pm\pi} = BD|_{\beta = \pm\pi} = CD|_{\beta = \pm\pi} = 0.$$

Thus, when the quaternion has an associated Euler angle β of zero, both A and B will also be zero. In that case atan(0/0) occurs in equation (5-9) and asin(0/0) occurs in equation (5-10). A similar situation happens when β is ±π, when the quaternion has values C = D = 0. We can then use the following scheme when relating quaternions to the Euler angles α and γ.

$$\alpha = \begin{cases} \frac{1}{2}\text{asin}\left(\frac{2CD}{C^2+D^2}\right) & \beta = 0 \\ \frac{1}{2}\text{asin}\left(\frac{-2AB}{A^2+B^2}\right) & \beta = \pm\pi \\ \text{atan}\left(\frac{AD+BC}{BD-AC}\right) & Else \end{cases} \qquad \gamma = \begin{cases} \frac{1}{2}\text{asin}\left(\frac{2CD}{C^2+D^2}\right) & \beta = 0 \\ \frac{1}{2}\text{asin}\left(\frac{2AB}{A^2+B^2}\right) & \beta = \pm\pi \\ \text{atan}\left(\frac{BC-AD}{AC+BD}\right) & Else \end{cases} \tag{5-11}$$

5.10.5 Multivalued Trigonometric Functions

As an example, consider the quaternion equivalent to the Euler angles α = γ = 0, β = –60°. From equations (5-1), the quaternion has A = C = 0, B = -0.5, and D = 0.866. Application of (5-5) produces the value β = -60° from the arcsine and β = 60° from the arccosine. This is shown by the black dots in the previous figure.

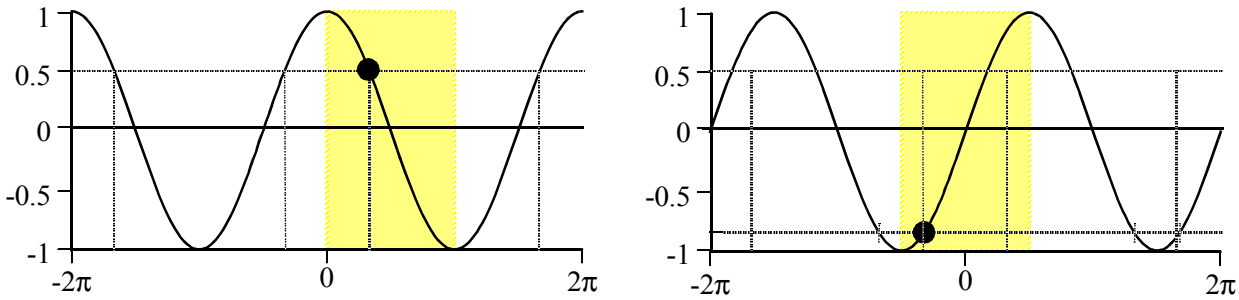


Figure A Sine and cosine functions such as those used to determine the angle β in a quaternion. The shaded regions indicate where the principle values are defined, i.e where the asin and acos functions are defined in ANSI C. For any value between [-1, 1] the arcsine (asin) and arccosine (acos) functions return angles that lie within the shaded regions shown. For both sine and cosine functions, dotted lines depict the possible angles equally valid for the input values (0.5 and -0.866 for cosine and sine respectively.)

Thus, from the equation involving the arcsine there are four angles possible, -150°, -60°, 210° and 300°. From the equation involving the arccosine there are also four possible angles, -300°, -60°, 60°, and 300°. By matching up the results from the arcsine and arccosine the possible values for β are seen to be either -60° or 300° degrees. These values correspond to equivalent Euler rotations, so either is valid.

5.10.6 Quaternion Equation Summary

Quaternion From Euler Angles

$$A = -\sin\frac{\beta}{2}\sin\frac{\alpha-\gamma}{2} \qquad B = \sin\frac{\beta}{2}\cos\frac{\alpha-\gamma}{2}$$

$$C = \cos\frac{\beta}{2}\sin\frac{\alpha+\gamma}{2} \qquad D = \cos\frac{\beta}{2}\cos\frac{\alpha+\gamma}{2}$$

Norm

$$A^2 + B^2 + C^2 + D^2 = 1$$

$$R = \begin{bmatrix} D_1 & -C_1 & B_1 & A_1 \\ C_1 & D_1 & -A_1 & B_1 \\ -B_1 & A_1 & D_1 & C_1 \\ -A_1 & -B_1 & -C_1 & D_1 \end{bmatrix} \qquad Q_1 Q_0 = R_1 Q_0 = \begin{bmatrix} A_0 D_1 - B_0 C_1 + C_0 B_1 + D_0 A_1 \\ A_0 C_1 + B_0 D_1 - C_0 A_1 + D_0 B_1 \\ -A_0 B_1 + B_0 A_1 + C_0 D_1 + D_0 C_1 \\ -A_0 A_1 - B_0 B_1 - C_0 C_1 + D_0 D_1 \end{bmatrix}$$

$$2\,\mathrm{asin}\left[\sqrt{A^2 + B^2}\right] = \beta = 2\,\mathrm{acos}\left[\sqrt{C^2 + D^2}\right]$$

5.11 Programs and Input Files

Lorentz0.cc

```
/* Lorentz0.cc ********************************************************************
**                                                                                  **
**              GAMMA Lorentzian Generation Program                                 **
**                                                                                  **
** This program uses the GAMMA Lorentzian function module to build and              **
** plot a simple Lorentzian function. The output will be interactive                **
** using Gnuplot. Both reals and imaginaries will be plotted.                       **
**                                                                                  **
** Author:    S.A. Smith                                                            **
** Date:      7/21/97                                                               **
** Copyright: S.A. Smith, September 1997                                            **
**                                                                                  **
**********************************************************************************/


#include <gamma.h>                              // Include GAMMA

main (int argc, char* argv[])
 {
 int qn=1;                                      // Query index
 int npts = 4096;                               // Set block size
 query_parameter(argc, argv, qn++,              // Ask for block size
        "\n\n\tNumber of Points? ", npts);

 double lwhh;                                   // For linewidth
 query_parameter(argc, argv, qn++,              // Ask for linewidth
     "\n\n\tHalf-height Linewidth? ", lwhh);
 double R = lwhh/2;                             // Set lwhh (2*R)

 double Wst, Wfi;
 query_parameter(argc, argv, qn++,              // Ask for start frequency
         "\n\n\tInitial Frequency? ", Wst);
 query_parameter(argc, argv, qn++,              // Ask for final frequency
         "\n\n\tFinal Frequency? ", Wfi);
 double W;                                      // The offset frequency
 query_parameter(argc, argv, qn++,              // Ask for offset
          "\n\n\tPeak Frequency? ", W);
 row_vector CL = Lorentzian(npts,Wst,Wfi,W,R);  // Here is Lorentzian
 GP_1D("CR.asc", CL, 0);                        // Lorentzian reals, GP ASCII
 GP_1Dplot("CR.gnu","CR.asc");                  // Plot reals of Lorentzian
 GP_1D("CI.asc", CL, -1);                       // Lorentzian imaginaries, GP
ASCII
 GP_1Dplot("CI.gnu","CI.asc");                  // Plot imaginaries of Lorentzian
 cout << "\n\n";                                // Keep the screen nice
 }
```

Lorentz1.cc

```
/* Lorentz1.cc ********************************************************************
**                                                                                  **
**        GAMMA Differential Lorentzian Generation Program                          **
**                                                                                  **
** This program uses the GAMMA Lorentzian function module to build and              **
** plot a simple differential Lorentzian function. The output will be               **
** interactive using Gnuplot. Both reals & imaginaries are plotted.                 **
**                                                                                  **
** Author:    S.A. Smith                                                            **
** Date:      7/21/97                                                               **
** Copyright: S.A. Smith, September 1997                                            **
**                                                                                  **
**********************************************************************************/


#include <gamma.h>                              // Include GAMMA

main (int argc, char* argv[])
 {
 int qn=1;                                      // Query index
 int npts = 4096;                               // Set block size
 query_parameter(argc, argv, qn++,              // Ask for block size
        "\n\n\tNumber of Points? ", npts);

 double lwhh;                                   // For linewidth
 query_parameter(argc, argv, qn++,              // Ask for linewidth
     "\n\n\tHalf-height Linewidth? ", lwhh);
 double R = lwhh/2;                             // Set lwhh (2*R)

 double Wst, Wfi;
 query_parameter(argc, argv, qn++,              // Ask for start frequency
         "\n\n\tInitial Frequency? ", Wst);
 query_parameter(argc, argv, qn++,              // Ask for final frequency
         "\n\n\tFinal Frequency? ", Wfi);
 double W;                                      // The offset frequency
 query_parameter(argc, argv, qn++,              // Ask for offset
          "\n\n\tPeak Frequency? ", W);
 row_vector CL = DLorentzian(npts,Wst,Wfi,W,R);// Here is differential Lorentzian
 GP_1D("CR.asc", CL, 0, Wst, Wfi);              // DLorentzian reals, GP ASCII
 GP_1Dplot("CR.gnu","CR.asc");                  // Plot reals of DLorentzian
 GP_1D("CI.asc", CL, -1, Wst, Wfi);             // DLorentzian imaginaries, GP
ASCII
 GP_1Dplot("CI.gnu","CI.asc");                  // Plot imaginaries of DLorentzian
 cout << "\n\n";                                // Keep the screen nice
 }
```