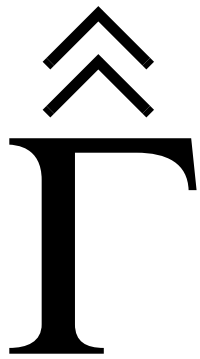
GAMMA

Users Manual



Author: Dr. Scott A. Smith, Tilo Levante

Date: October 21, 1999

Users Guide

Chapters

1 Introduction

THIS DOCUMENT ASSUMES THAT THE READER HAS A WORKING VERSION OF Γ.

GAMMA is a computational environment designed to simplify the simulation of magnetic resonance (MR) phenomena. It accomplishes this by allowing users to write C++ programs using the objects that are commonly used to describe MR. Thus GAMMA programs are C++ programs but they make use of quantities such as spin systems, pulses, delays, Hamiltonians, and the like. This document describes the basics of using GAMMA and writing GAMMA programs. It will be biased towards simulations of nuclear magnetic resonance (NMR) problems as well as towards use in a Unix environment. It also assumes that the reader is familiar with some editor and that the GAMMA platform has been successfully installed.

1.1 C++ Program Basics

The purpose of GAMMA is to allow you, the user, to write your own programs easily and efficiently. In this regard it is similar to programs such as MATLAB, Mathematica, MAPLE, etc. However GAMMA programs **are** C++ programs, and that implies that

1 You have a full computer language at your disposal with all its flexibility & added libraries.

2 You will have to gain some rudimentary knowledge of programming[1] in C++.

There is a learning curve associated with GAMMA but we hope you will find that small relative to what you will gain in ability.

1.1.1 Basic C++ Program: Hello Cruel World

We begin by writing a “GAMMA” (actually a C++) program. A simple program would be the following:

```
#include <gamma.h>
main()
  {
  cout << “\n\tHello World\n”;
  }
```

The first line is a compiler directive to include the GAMMA platform[2]. You can have several lines such as this to include more files as desired. In this program GAMMA isn’t providing anything so the line could be left out.

The next line declares that the main part of the program is following, enclosed by the brackets {.....}. These brackets must both be present and any code associated with main must reside be-

1. The author was a steadfast FORTRAN user until the GAMMA project began. Some can argue that he still doesn’t know how to program in C (and barely manages in C++), but he now avoids FORTRAN unless its shoved in his face. Given that a slow learner such as him can manage in C++, you should be able to as well.
2. Actually, this line inlcudes the file gamma.h which, in turn, contains all of the GAMMA source headers.

tween them.

The only line of code is just to write “Hello World” to standard output. A few things to note.

1 Standard output (your screen by default) is called **cout**.

2 The **operator** << is one that the object itself (the stuff enclosed in ““, a String) knows all about.

3 The special combinations, **\n** and **\t**, are a page break and tab respectively when in a String.

4 Every code line must end with a semicolon, **; (except things enclosed in {})**.

Now we must convert this program code into an executable. That is done by using the “gamma” script which invokes the C++ compiler, links any C++ libraries and links to the GAMMA platform. We sill use **noesy>** to be the prompt that the computer we are working on uses, and we will assume the program is called hello.cc (note that the .cc is often mandatory when using the C++ compiler!). Here we go.

```
noesy> gamma hello.cc
                    G A M M A
                [Other Messages]

noesy>
```

Your actual response may vary depending on your GAMMA version and your computer type. The command **gamma** acting on hello.cc (or any C++ program) will produce an executable that has a default name[1] called **a.out**. To run the program,

```
noesy> a.out
                    Hello World

noesy>
```

That’s it. If this worked you have completed compilation of your first C++ program and future compilation your GAMMA programs will be very similar.

1.1.2 More C++ Basics: Comments, Spacing, Executables

We’ll will cover a few of the basics of C++ programming before using GAMMA for anything interesting. More key points to remember:

1 Blank spaces are ignored, it doesn’t matter if there is 1 or 20 or none. Thus, there is no column alignment of code, code may span multiple lines (i.e. multiple lines before a “;”), or there may be multiple “;’s” on a single line.

2 Comments in C++ may be either in the typical C fashion (i.e. anything between /* and */ is taken as a comment) or by use of a // (anything past // on that line is a comment).

3 The output executable can be named anything you like by using “-o outfile” during compi-

1. On Intel based PC’s running Windows the executable file is a.exe rather than a.out.

lation of your program.

Now, we can make some simple modifications and repeat the compilation process. Let me adjust the program would be the following:

```
#include <gamma.h>                    // Include the GAMMA library
/*******************************
** Here's the same program again **
*******************************/
main()
  {  cout << "\n\tHello World\n";  }   // Output the message
```

Assuming that this file is named hello1.cc, we shall compile and name the output executable "again".

```
noesy> gamma hello1.cc -o again
                    G  A  M  M  A
                [Other Messages]

noesy>
```

To run the program,

```
noesy> again
                    Hello World

noesy>
```

Same result as before, I just wanted to demo how comments are included and how the executables can be named.

1.1.3 Basic C++ Program: Includes, Subroutines, Running Interactive

Three more basic areas and then we'll do something with GAMMA.

1 Include statements are used to tell the C++ compiler about specific files it should know . These may be **header files (filename.h),** files that indicate how to interface with various functions and data types. Typically a header file will have associated code that has already been compiled and will be included during program compilation either in a library or an object file. Include files can also be C++ code **(filename.cc),** in which case the code is also compiled when the program is compiled. Include statements begin with a # and the filename will be encased either with brackets to indicate it is in a searched directory (e.g. # include <filename.h>) or in quotes indicating it is in the local directory (e.g. # include "filename.cc").

2 Subroutines, or functions, which are in the file containing the main source code will normally reside after the include statements but before the main program. The main program will see only routines which are above it in the file. Functions and subroutine may also exist in separate files, included as indicated in the previous paragraph.

3 Interactive programs may be written either by allowing the main "function" to accept arguments or by use of standard input.

Our simple programs have already used the “#include” statement so that they have been able to use GAMMA (but haven’t). If you wish to include other programs and modules you will have to experiment, that’s really beyond the level of this section. However, here is a small modification that uses a function. The function precedes the main part of the program but is included in the same file.

```
#include <gamma.h>

string evenworse(int i)
  {
  if(i<=0) return string(“Cruel”);
  else if(i==1) return string(“Very Cruel”);
  return string(“Extremely Cruel”);
  }

main()
  {
  cout << “\n\tJust How Cruel [0,1,2]? “;
  int i=0;
  cin >> i;
  cout << “\n\tHello “ << evenworse(i) << “ World\n”;
  }
```

Assuming that this file is named hello2.cc, we shall compile and name the output executable “onemore”.

```
noesy> gamma hello2.cc -o onemore
                    G  A  M  M  A
                [Other Messages]

noesy>
```

To run the program,

```
noesy> onemore

          Just How Cruel [0,1,2]? 2
        Hello Extremely Cruel World

noesy>
```

We could have added more functions that take different arguments and we could have put the function codes in external files. You’ll have to learn as you go. This covers the very basics of C++ programming. If you wish to become versed in C++ buy a nice book on that subject and look at other programs. The reset of this document will teach you some C++ as we make GAMMA programs and you can look at the GAMMA sources and example programs for other ways to do things.

2 Classes

In the previous chapter of this document you learned the basics of constructing and compiling simple C++ programs. In this chapter we shall start writing simple programs which use the objects which are defined both in C++ and in GAMMA. The programs herein will highlight some of the features of using GAMMA classes, BUT to take full advantage of them you should look at the GAMMA Modules Documentation . Each module contains a set of associated files, and these files define programming objects and functions. There will be a chapter for each GAMMA class type and a full list of the functions and operators defined for them.

2.1 Double Values

Consider the data type **double**, intrinsic in C++. Such variables are used to track floating point numbers with double precision accuracy. We bring this up here so that those who are new to C++ can see how to utilize double precision numbers in their programs. Subsequently, data types supplied by GAMMA will be used with similar lines of code.

Variables of type **double** have the following **data type** properties:

1 A double precision number may be declared anywhere in a program.

2 An array of double precision numbers can be readily declared and accessed with [].

3 Double precision numbers come with their own functions (exp, <<).

4 Double precision numbers have a defined algebra (+, *, /, ...).

5 Double precision number have the ability to interact with other data types (int + double).

Other intrinsic data types - such as integers, strings - have similar characteristics. In GAMMA still more data types are defined and also have such features (matrices, operators, tensors...). In the next sections we'll look at some simple programs with double, then simple programs using GAMMA defined (non-intrinsic) data types.

2.1.1 Basic C++ Program Using Doubles

We begin by writing a "GAMMA" (actually a C++) program. A simple program would be:

```
#include <gamma.h>
main()
  {
  double x;                                  // This is an empty double
  double y = 5.0;                            // This is double y of value 5
  cout << "\nValue y is " << y << "\n";       // Print value of y to standard out
  x = 12.4/y + exp(1.07);                    // Set double x to some value
  xarr[10];                                  // This is an array of doubles
  for(int i=0; i<10; i++)                    // A loop to fill up xarr
  xarr[i] = x - i*y;
  cout << "\nFirst array value:  " << xarr[0];   // Print the first array value
  cout << "\nLast array value:  " << xarr[9];    // Print the last array value
```

```
double tanx = tan(x);                    // Here is the tangent of x
}
```

Again we'll emphasize what the intrinsic class **double** means:

1. All doubles have a well defined interface. Not to much worry about multiplying and adding them, one knows exactly what to expect.
2. They have a set of functions which apply to them and perhaps to other types as well. Thus the **operator <<** is known to both doubles and strings, the operator log will be known to both integers and doubles, and so on.

Now we must convert this program code into an executable. That is done by using the "gamma" script which invokes the C++ compiler, links any required C++ libraries, and links to the GAMMA platform. We still use **noesy>** to be the prompt that the computer we are working on uses, and we will assume the file containing this program code is called dbl.cc (note that the .cc is often mandatory when using the C++ compiler!). Here we go.

```
noesy> gamma dbl.cc

        G  A  M  M  A

    [Other Messages]

noesy>
```

Your actual response may vary depending on your GAMMA version and your computer type. The command **gamma** acting on dbl.cc (or any C++ program) will produce an executable that has a default name called **a.out**. To run the program,

```
noesy> a.out

    [Output From Program]

noesy>
```

I won't include the output of this run, the program isn't meant to do anything constructive. It is only to show what a simple C++ program using doubles will look like,

2.2 Matrices and Vectors

Let us now jump up a level in abstraction yet remain focused on mathematical manipulations. GAMMA provides data types **matrix**, **row_vector**, and **col_vector** to handle matrices, row vectors and column vectors respectively. What does that mean? It means that you are free to manipulate these objects in your GAMMA programs just as you freely manipulated double precision numbers

in the last example.

Variables of type **matrix, row_vector, and col_vector** have the following **data type** properties:

1 Matrices and vectors may be declared anywhere in a program.

2 An array of matrices and or vectors can be readily declared and accessed with [].

3 Matrices and Vectors come with their own functions (exp, <<).

4 Matrices and Vectors have a defined algebra (+, *, ...).

2.2.1 Basic GAMMA Program Using Matrices & Vectors

Have a look at the following code.

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

  double x;                              // This is an empty double
  matrix mx;                             // This is an empty matrix
  row_vector rv;                         // This is an empty row vector
  col_vector cvs[10];                    // These are 10 empty column vectors
  matrix mx1(2,3, 7);                    // A 2x3 matrix filled with 7's
  complex z(2,-1.3);                     // A complex number 2-1.3i
  matrix mx2(3,5,z);                     // A3x5 matrix filled with z
  mx2.put(complex(2,2),0,1);             // Set <1|mx|2> to be 2+2i
  rv = row_vector(3,-1);                 // Now rv's a row vector of length 3 with -1's
  matrix mx3 = exp(x)*rv*mx2/complexi;// What the heck, just playing around.
  cout << mx3;                           // Let's have a look at mx3....isn't it 1x5?
  }
```

In GAMMA programs you can build up any vectors and matrices you need and then you may manipulate them as readily as you would a double precision number! Make arrays of matrices, take their exponentials, do whatever you like within reason... they are objects for you to wield to your hearts content, much in the same way you can do in MATLAB.

I'll no longer bother with the compilation step, just look to the previous examples. Here is the result of running the above program:

```
GAMMA 1 x 5 Full Matrix


(  3.90,  6.00) (  0.60,  6.00) (  3.90,  6.00) (  3.90,  6.00) (  3.90,  6.00)
```

Note that the strength of GAMMA in NOT in it matrix and vector manipulations! That is a powerful feature, but it is shared by other types of programs to some extent. (Yes, GAMMA can read and write matrices to and from MATLAB...) The wonderful thing about GAMMA for those working in magnetic resonance will be demonstrated through use of the MR tailored classes.

2.3 Basic Spin Systems

At this point we will depart from the “mathematical” classes and switch our focus to a GAMMA provided class **spin_sys**. This is a data type which embodies a fundamental entity in magnetic resonance, namely a collection of spins and associated spin quantum numbers. I will not continue emphasizing the flexibility one has when working with data types. (Yes you can make an array of spin systems if you wish).

2.3.1 Base Spin System: Primitive Construction, Standard Output

Heres a simple program which is very much like the original “Hello World” program.

```
#include <gamma.h>
main()
  {
  spin_sys sys(3);                                    // System “sys”, 3 spins
  cout << “\n\tA Default Three Spin System\n” << sys;// Write sys to standard output
  }
```

The program just declares a spin system and then the system writes itself to standard output. We’ll compile and run the program, taking sys1.cc to be the name of the file containing the above code.

```
noesy> gamma sysl.cc

                G  A  M  M  A

             [Other Messages]

noesy> a.out

       A Default Three Spin System
System   :
Spin     :    0         1         2
Isotope  :   1H        1H        1H
Momentum :  1/2       1/2       1/2

noesy>
```

Because we have not specified any details other than that the system contains 3 spins, GAMMA automatically uses a default isotope of 1H. Also, note that there is a standard output function, “<<“ defined for a spin system. In effect, the system knows how to display itself to the screen (as do doubles, integers, matrices, and most of the data types in GAMMA).

2.3.2 Base System: Member Functions, Info Access, File Input

Now we’ll get a bit more fancy with our basic spin system. Since in C++ we have control over what functions are available to these type of variables, lets consider some that might be useful.

1 Set./Retrieve the number of spins

2 Set/Retrieve a spin's particular isotope type

3 Read/Write spin system to disk

4 Get the system Hilbert space dimension.

5 Obtains a spins angular momentum and/or gyromagnetic ratio

6 .....

All of these are **intrinsic** properties of any spin system and therefore available to GAMMA programs at any time. Have a look at the following variation of the previous program. I'm gonna make it more sophisticated now that you know a bit of C++ and GAMMA....

```
#include <gamma.h>
main()
  {
  spin_sys sys;                                  // System "sys"
  sys.read("test.sys");                          // System reads itself from file test.sys
  cout << sys;                                   // Have a look at the system.
  cout << "\n2nd spin Iz: " << sys.qn(1);        // Here is Iz of the 2nd spin
  cout << "\n1st spin is " << sys.element(0);    // This is the 1st spin type
  cout << "\nSystem Hilbert space is " << sys.HS();// This is the spin Hilbert space
  if(sys.homonuclear())                          // Tell us if it homo/hetero nuclear
    cout << "\nSystem is Homonuclear";
  else
      cout << "\nSystem is Heteronuclear";
  }
```

The above code may seem cryptic to those used C and FORTRAN programs because it makes use of **member functions**. Rather that making use of the function sin on the variable x via sin(x), the member function use of sine might be written x.sin(), i.e. the function is attached to the data type by a single period ".". In the above code, the spin_sys member functions read, qn, element, HS, and homonuclear are used. It'll take some getting used to but once you familiarize yourself with this syntax you'll start to like it. However, there may be both member and non-member functions defined for a particular data type - but that is simple - you just need to look up the function and its usage.

Pay particular attention to the fact that this program DOES NOT contain any information about the spin system, it is system independent (although I do ask for information on the 1st two spins, so the program would give an error if the system doesn't have at least two spins...).

When the program is run it will look for a file "test.sys" that contains information that defines the system. Here is an example of such a file:

This is a Example of A File Containing A Basic GAMMA Spin System

```
SysName     (2) : CDV      - Name of the Spin System
NSpins      (0) : 3        - Number of Spins in the System
Iso(0)      (2) : 13C      - Spin Isotope Type
Iso(1)      (2) : 2H       - Spin Isotope Type
Iso(2)      (2) : 51V      - Spin Isotope Type
```

If I compile the program and the above system information is in a file "test.sys" here will be the

program output:

```
System    : CDV
Spin      :   0         1         2
Isotope   : 13C        2H       51V
Momentum  : 1/2         1       7/2

2nd spin Iz: 1
1st spin is C
System Hilbert space is 48
```

If you use a different file you will of course get different results.

2.3.3 GAMMA Base Spin System: Interactive

Suppose now that you like the above program very much but, rather than having it always read the file “test.sys” to get the spin system you would like it to ask you for which file to read the system from. That can be done crudely by use of code such as

```
spin_sys sys;                          // System “sys”
string filename;                       // A string for the filename
cout << “\n\n\tWhich file? “;           // Ask the user for a filename
cin >> filename;                       // Get the filename from user
sys.read(filename);                    // System reads itself from file test.sys
```

substituted in the previous program. I’ll use a more sophisticated approach because its something that is nice to use once you take the time to learn it. Here’s a rewrite of the previous program start:

```
#include <gamma.h>
main(int argc, char *argv[])
  {
  spin_sys sys;                        // System “sys”
  sys.ask_read(argc, argv, 1);         // Ask for and/or Read system
  cout << sys;                         // Have a look at the system.
  //.....                              // Rest of the program here!
  }
```

The key concepts here are 1.) The change to the main program to take an integer and an array of strings and 2.) Use of the member function ask_read to have the spin system ask the user which file it should read itself from.

The former is standard in C and C++ - you don’t need to understand it, you can just always write you “main” program statement in such a manner. What that does is provide the program with argc, the number of arguments given on the command line when the program is run, and argv, the arguments given on the command line.

The latter is just part of class spin_sys in GAMMA. If the 1st argument (via the 1 in the call) is provided when the program is executed then sys will use that value as the filename it should use to read itself. If no 1st argument is provided then the system will ask the user for a filename.

Don’t spend too much time worrying about the details here. You’ll learn this stuff with experience. Below is the above program run both with and without a spin system file name on the command line. I’m leaving out the code following cout << sys for brevity. Here is the program (executable

named a.out) run when the name test.sys is supplied on the command line:

```
System    : CDV
Spin      :          0          1          2
Isotope   :        13C         2H        51V
Momentum  :        1/2          1        7/2
```

Here is the same program run when no arguments are supplied on the command line:

```
|gamma1>a.out

Spin system filename? test.sys
System   : CDV
Spin     :    0          1          2
Isotope  :  13C         2H        51V
Momentum :  1/2          1        7/2
```

See the difference? Now your GAMMA program can be run repeatedly with any number of input spin system files. Yep, the current program doesn't do much.... but wait until you use spin systems to do simulations later. This is the means by which you will soon learn how one can just make a general COSY simulator for any spin system (containing any isotopes!). The above was run on my SPARC20, a machine with the prompt |gamma1> so don't let that worry you.

GAMMA is not limited to the number of spins in a spin system and has a internal knowledge of most spin isotopes! To learn more about GAMMA's spin system(s) and isotopes see their chapters in the GAMMA module documentation (Basics & HSLib).

2.4 Spin Operators

Having learned about basic spin systems, we shall start learning to use something which is fundamental to the mathematical treatment of magnetic resonance, a spin operator. GAMMA provides the user with a wide variety of functions that return spin operators - operators based on spin angular momentum - that reside in a composite spin space. These functions almost invariably take a GAMMA spin system as a function argument.

2.4.1 Spin Ops: Construction, Functions, Output

To keep things simple, the following program will just read in the system file (test.sys) rather than ask the user for it.

```
#include <gamma.h>
main()
 {
 spin_sys sys;                                  // System "sys"
 sys.read("test.sys");                          // System reads itself from file test.sys
 spin_op FZ = Fz(sys);                          // Here is Fz for the system
 cout << "\nSystem Total Fz Op: " << FZ;         // Have a look at Fz for the system
 cout << "\nSystem F+: "                        // Here is F+ for the system output
 << Fx(sys)+complexi*Fy(sys);                   // to the screen
 //cout << "\nSystem F+: " << Fp(sys);          // This is also F+ by a easier way
 double dij = 134.7                             // Dipolar coupling value
```

```
gen_op HD = dij*Fz(sys,0)*Fz(sys,1);                    // Dipolar Hamiltonian component}
}
```

Since, according to the previous program demonstrating class spin_sys, the Hilbert space is 34 for the “test.sys” defined spin system, all our output operators will be 34x34 arrays, too big to do a screen capture and have on this page. So, I’m going to use a smaller spin system defined in my “test.sys”. Here is the one I will use instead (just tritium instead of vanadium)

```
SysName     (2) : 3Spins    - Name of the Spin System
NSpins      (0) : 3         - Number of Spins in the System
Iso(0)      (2) : 13C       - Spin Isotope Type
Iso(1)      (2) : 2H        - Spin Isotope Type
Iso(2)      (2) : 3H        - Spin Isotope Type
```

I’m also going to use a GAMMA FrameMaker function so I can bring them right into this document. Rather than use the function cout << spin_op (as is shown in the above program) I will substitute in FM_Matrix(“file.mif”, spin_op); Those of you who don’t use FrameMaker don’t need to worry about this, suffice it to say that the following matrices are one in the same as the ones that would appear on screen if you ran the program with the 3-spin CDV system used previously (except they would show up on screen as diagonal and Hermitian arrays...)

$$Fz = \begin{bmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -2 \end{bmatrix} \qquad F+ = \begin{bmatrix} 0 & 1 & 1.41 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1.41 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1.41 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1.41 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1.41 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.41 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1.41 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1.41 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Note also that I cheated on the last line of the program and used GAMMA class gen_op, the general quantum mechanical operator class. We’ll get back to that later, I just wanted to show those who know the math how one can build up various Hamiltonians. Of course, there are functions in GAMMA for doing such things in 1 step... but you can build up ANY spin Hamiltonians you like and manipulate them in whatever way you need.

To find out which spin operators are available by simple function calls see the GAMMA ***MR Library Documentation***. You’ll find that all commonly used spin operators are there including spin rotation operators. Users can build up any such operators if there is no function to do so. To see how GAMMA spin operators are constructed and their functionality see the GAMMA ***Class Documentation***.

2.5 Isotropic Spin Systems

The last two sections have shown how basic spin systems (variables of class spin_sys) are defined and, in turn, are used to produce spin operators in a completely generalized manner. Spin systems are "containers" of information about the spin isotopes in a sample. The system provides that information to the spin operator functions and this sets the stage for building spin Hamiltonians and ultimately for applying pulses, delays, and acquisitions to the sample.

Consider the high-resolution isotropic NMR Hamiltonian. It is given by

$$\boldsymbol{H}_o = \sum_{i=1}^{spins} -\omega_i \boldsymbol{I}_{iz} + \sum_{i}^{spins} \sum_{i>i}^{spins} J_{ij} \boldsymbol{I}_i \bullet \boldsymbol{I}_j ,$$

We have already seen that we can readily make $\boldsymbol{I}_{iz}$ spin operators for each spin in the system. A line of code might be (where sys is the spin system)

```
spin_op Iz0 = Fz(sys,0)                    // Iz for the 1st spin
```

Without much thought ($\boldsymbol{I}_i \bullet \boldsymbol{I}_j = \boldsymbol{I}_{iz}\boldsymbol{I}_{jz} + \boldsymbol{I}_{ix}\boldsymbol{I}_{jx} + \boldsymbol{I}_{iy}\boldsymbol{I}_{jy}$) we know we can readily make the spin operators required for scalar coupled spin pairs. Here is a code line that would partially suffice

```
spin_op I0I1 = Fz(sys,0)*Fz(sys,1)         // Iz1*Iz2 for the 1st spin pair
             + Fx(sys,0)*Fx(sys,1)
             + Fy(sys,0)*Fy(sys,1)
```

If we stretch our imaginations we can just put in a loop over the number of spins and the number of spin pairs and do a summation. The code C++ code would look like

```
int i,j;
for(i=0; i<sys.spins(); i++)
 {
 // Add in the chemical shift contributions here
 for(j=i+1; j<sys.spins(); j++)
  {
  // Add in the scalar coupling contributions here
  }
 }
```

However we still lack some important information, particularly the chemical shifts of all the system spins and the scalar couplings between the system spin pairs. One solution would be to just add this isotropic information directly into the spin system and let the system itself tell us what these values are. That is (almost) exactly what GAMMA does. However, it does NOT use the basic spin system class, **spin_sys**, to do so. It uses an isotropic spin system class, **spin_system**, to do that job. Variables of class **spin_system** contain all of the information that variables of class **spin_sys** do but in addition they contain isotropic shift values for each spin and isotropic scalar couplings for each spin pair!

Now, lets have a look at building our isotropic Hamiltonian again.

```
spin_system sys;                        // An isotropic system (not spin_sys!)
sys.read("ABX.sys");                    // Read in the system from file
int i, j, ns=sys.spins();               // Needed integers, number of spins
for(i=0; i<ns; i++)                     // Loop over the spins
 {
 H0 -= sys.shift(i)*Fz(sys,i);          //   Add shift contributions
 for(j=i+1; j<ns; j++)                  //    Loop spin pairs
  {
  H0 += sys.J(i,j)                      //   Add coupling contributions
  * ( Fz(sys,i)*Fz(sys,j)
  + Fx(sys,i)*Fx(sys,j)
  + Fy(sys,i)*Fy(sys,j) );
  }
 }
```

This is NOT a complete GAMMA program (I haven't defined H0 yet). However it does illustrate a key concept in C++: derived classes. GAMMA's class **spin_system** is derived from the base spin system class **spin_sys**. As such, all functions that take variables of type **spin_sy**s will also take variables of type **spin_system**. So, accessing the spin operators in the above code looks identical to the previous programs which used base systems - except now we are putting in an isotropic spin system when calling the functions.

2.5.1 Isotropic Spin System: Interactive, NMR Hamiltonian

To illustrate this, lets now make the isotropic NMR Hamiltonian. We'll read in a spin system from an external file and then build and output the isotropic Hamiltonian to the screen. Here goes:

```
#include <gamma.h>
main(int argc, char *argv[])
 {
 spin_system sys;                       // System "sys"
 sys.ask_read(argc, argv, 1);           // Ask for and/or Read system
 cout << Ho(sys);                       // Have a look at Ho.
 }
```

Well? Not too hard, was it? Hopefully you haven't forgotten about the "ask_read" function discussed in the **spin_sys** section, nor the arguments in the "main" call. Sure, I could have left in the looping over spins and spin pairs but the isotropic Hamiltonian is simply used too often in NMR simulations. Thus, it is just a function in GAMMA (that we will learn all about shortly). If you looked at the code for the "Ho" function you would find that it is just that, a loop over the spins and spin pairs and the summing up of components. Lets try out the program. I'll use the following ASCII file, "sosi.sys", as my input system

```
SysName     (2) : C-D       - Name of the Spin System
NSpins      (0) : 2         - Number of Spins in the System
Iso(0)      (2) : 13C       - Spin Isotope Type
Iso(1)      (2) : 2H        - Spin Isotope Type
J(0,1)      (1) : 22.0      - Scalar coupling (Hz)
v(0)        (1) : 1200      - Shift of spin 1 (Hz)
PPM(1)      (1) : 0         - Shift of spin 2 (PPM)
Omega       (1) : 600.00    - Spec. Freq. in MHz (1H based)
```

Remember, you can have any number of spins in a spin system and virtually any spin isotope. Now I'll run the program with this file.

```
|gamma1>a.out sosi.sys
```

```
Matrix:
GAMMA 6 x 6 Diagonal Matrix

(-589.00,  0.00)
      (-600.00,  0.00)
            (-611.00,  0.00)
                  (589.00,  0.00)
                        (600.00,  0.00)
                              (611.00, -0.00)
Basis:
Default Basis (6 x 6) Identity Matrix
```

O.K., that was pretty uneventful. The Hamiltonian is diagonal because the scalar coupling in heteronuclear. The GAMMA function Ho knows that from the system and automatically sets weak scalar coupling. Let me rerun after switching the deuterium to carbon (replacing 2H by 13C in the file sosi.sys). Now here is the output:

```
|gamma1>a.out sosi.sys
Matrix:
GAMMA 4 x 4 Full Matrix

(-594.50,  0.00) (  0.00,  0.00)     (  0.00,  0.00)    (  0.00,  0.00)
(  0.00,  0.00)     (-605.50,  0.00) ( 11.00,  0.00)   (  0.00,  0.00)
(  0.00,  0.00)     ( 11.00,  0.00)    (594.50,  0.00)  (  0.00,  0.00)
(  0.00,  0.00)     (  0.00,  0.00)     (  0.00,  0.00)     (605.50,  0.00)
Basis:
Default Basis (4 x 4) Identity Matrix
```

Still dull, but at least we have off-diagonals! Note how, although the GAMMA program has nothing specific about the input system, the output Hamiltonian automatically adjusts depending upon the system used. We will later deal with Hamiltonians and operators, operators being the return data type from function Ho - that is why the output talks about a basis.

2.5.2 GAMMA Isotropic Spin System: Access Functions

Now you know about isotropic systems, how they do everything base spin systems do, and how readily they may be used in GAMMA programs. Now think how you might run a program that loops through a set of similar spin systems, perhaps fitting some results to spin system parameters or watching the effects of strong coupling a two spins shifts mover closer together.

To do that you'll need full access to the spin system information, i.e. be able to set chemical shifts, coupling constants, isotope types, etc. within your GAMMA programs. Not a problem. The next program illustrates a couple of these abilities.

```
#include <gamma.h>
main(int argc, char *argv[])
  {
  spin_system sys(4);                    // A system of 4spins (all 1H!)
  cout << sys;                           // Lets have a look at it
  sys.isotope(2,"31P");                  // Set the 3rd spin to phosphorous
  sys.Omega(900.0);                      // Set field for 900 MHz proton (yeah)
  sys.PPM(7.2, 0);                       // Set 1st spin shift to 7.2 PPM
```

```
sys.J(0, 1, 11.0);                    // Set J12 to be 11 Hz
cout << sys;                          // Lets have another look
}
```

Here's the output from this little ditty:

```
a.out sosi.sys

Spin            :         0         1         2         3
Isotope         :        1H        1H        1H        1H
Momentum        :       1/2       1/2       1/2       1/2

Shifts          :      0.00      0.00      0.00      0.00
J Values (Hz)
Spin  0         :                0.00      0.00      0.00
Spin  1         :                          0.00      0.00
Spin  2         :                                    0.00

Spin            :         0         1         2         3
Isotope         :        1H        1H       31P        1H
Momentum        :       1/2       1/2       1/2       1/2

Shifts          :    6.48 K      0.00      0.00      0.00
PPM             :      7.20      0.00      0.00      0.00
J Values (Hz)
Spin  0         :               11.00      0.00      0.00
Spin  1         :                          0.00      0.00
Spin  2         :                                    0.00
Omega           :    900.00 M  900.00 M  364.33 M  900.00 M
```

Ho hum....Lets move on, you'll get your fill of spin systems as you go through this document. See the GAMMA CLASS DOCUMENTATION for all of the spin_system functions and what parameters are important in external ASCII files that may be used to define them.

2.6 General Operators

Often, magnetic resonance problems are described in terms of operators: whether spin operators (as we have already seen), product operators, single transition operators, even Hamiltonian operators. To handle generalized quantum mechanical operators GAMMA contains class gen_op. In the same way that GAMMA contains a battery of function to construct common spin operators, GAMMA contains a variety of functions that construct common general operators. An example of that was the function Ho that was used in an earlier program.

2.6.1 Operators: Hamiltonians, Propagators, Density Operators

To demonstrate the use of GAMMA operators we'll approach a simple equation often encountered in NMR, the evolution of the spin system during a delay under a constant Hamiltonian. The spin

system will be embodied by a density operator and we'll use the isotropic NMR Hamiltonian for our constant Hamiltonian. Here is the math (solution to the Liouville equation under constant H):

$$\sigma(t) = e^{-iHt}\sigma_o e^{iHt} \tag{2-1}$$

Let's try and implement such an evolution in a program. Here goes:

```
#include <gamma.h>
main(int argc, char *argv[])
 {
 spin_system sys;                          // System "sys"
 sys.ask_read(argc, argv, 1);              // Ask for and/or Read system
 gen_op sigma = Fx(sys);                   // Start with pure x magnetization
 gen_op H = Ho(sys);                       // Have a look at Ho
 double t = 1.23;                          // Set time for 1.23 seconds
 gen_op sigma1 = evolve(sigma, H, t);      // Evolve sigma under H for time t
 }
```

You are very close to a 1D NMR simulation. In this example the operator sigma is used to represent the state of the spin system following a perfect 90y pulse (pure X magnetization). The operator H is set to the isotropic NMR Hamiltonian and then a new density operator, sigma1, is made by evolving sigma under the Hamiltonian H for a time t using the function "evolve".

That might be too cryptic for some, since the function evolve does hide the underlying mathematics just as the functions Ho and Fx hide the mathematics behind themselves. But that is only for convenience, there is nothing preventing the user from doing the step by step processes explicitly. Here is another way to do the evolution in the last line of the previous code:

```
gen_op U = exp(-complexi*2.0*PI*H*t);
gen_op sigma1 = U*sigma*adjoint(U);
```

Here's another way to do it:

```
gen_op U = exp(-complexi*2.0*PI*H*t);
gen_op sigma1 = evolve(sigma,U);
```

Here's yet another way to do it:

```
gen_op U = prop(H, t);
gen_op sigma1 = evolve(sigma,U);
```

We could make a few dozen more too. As you get used to using GAMMA and become convinced that functions such as Ho are every bit as good as you writing out the sums over spin operators you'll switch to the simpler (more cryptic? not really, Ho is H_o and evolve is just that...) code. But if you don't like to, write out the steps. Sometimes one needs a specialized Hamiltonian or operators and the best way to get it is to just explicitly add up various spin components!

3 Magnetic Resonance

At this point you have learned how to build C++ and GAMMA programs. You've also experienced the use of GAMMA provided data types: *spin systems, operators, superoperators, tensors, etc*. Along with these data types, GAMMA also provides a number of functions which perform manipulations on them which are common to magnetic resonance simulations.

For example, rather than building the isotropic NMR Hamiltonian stepwise there is a simple function which returns that Hamiltonian (as an operator, *gen_op*) in a single step. Rather than writing a subroutine to make a dipolar relaxation matrix there is a simple function which will return it (as a superoperator, *super_op*.) Rather than looping over the points of an acquisition there are functions which will fill a data block with the FID points. And so on....

You don't have to use these functions in your GAMMA programs just as you are not required to use GAMMA data types in your C++ programs. But the idea is that, once you are used to them, you can focus your programming efforts on new and exciting things rather than spending a week to program in a 90 pulse.....

3.1 Hamiltonians

This document already made use of the function which provides the isotropic Hamiltonian, **Ho**. It is just a sum of the isotropic shift Hamiltonian and the isotropic scalar coupling Hamiltonian. That's all we'd ever need for NMR simulations if we always dealt with small molecules in nicely liquid systems where relaxation didn't concern us.

But the fact is that we do deal strongly relaxting systems, large molecules, powder samples, liquid crystals, exchanging systems, etc. If we want to do simulations on those (we do, we do...) then we'll need some other Hamiltonians at our disposal. Remember, you can just build any Hamiltonian you wish by adding and multiplying spin operators. The Hamiltonian "functions" provided are just the ones so common that we don't want to have to think about them.

3.1.1 Isotropic NMR

First we'll make some isotropic ones (good for liquid NMR simulations). Remember, the key here is to use a "spin_system", a spin system that internally knows about isotropic shifts, isotropic scalar couplings, isotope types, gyromagnetic ratios, ...

```
#include <gamma.h>
main()
  {
  spin_system sys;                    // Here is a spin system, empty though
  sys.read("ABMX.sys");               // Set the system from ASCII file ABMX.sys
  gen_op H = Ho(sys);                 // Our friend, the isotropic NMR Ham.
  gen_op H_S = Hcs(sys);              // Just the shift part of Ho
  gen_op H_SL = Hcs_lab(sys);         // Shift in the lab frame (big #'s here!)
  gen_op H_J = HJ(sys);               // Isotropic scalar coupling (STRONG!)
  gen_op H_JW = HJw(sys);             // Isotrpic scalar coupling (WEAK!)
```

```
gen_op H_JWH = HJwh(sys);              // Isotropic scalar coupling (Weak Hetero!)
gen_op H1 = H_S + H_JWH;               // Same as H above!
gen_op HZ = Hz(sys);                   // Zeeman Hamiltonian (big #'s if in a field)
gen_op HZI = Hz(sys, "2H");            // Zeeman Hamiltonian for any deuteriums
gen_op HQ = HQsec(sys, 2.e6, 0);       // Quad. Ham., wQ 2 MHz, 1st spin
}
```

We could go on but I think you might be a bored as I am. Have a look in the GAMMA MR Library Documentation for all the Hamiltonian functions. The really important part of all of this is only that you have simple likes of code to get some need Hamiltonians. Even better, you can manipulate the Hamiltonians because they are just operators (gen_op).

Note: For Anisotropic Hamitonians, See Spatial & Spin Tensors and the Rank 2 Interactions

3.2 Ideal Pulses

This the the stuff to know about if you don't care about artifacts from pulse offsets, pulse power, and pulse lengths. Ideal pulses are perfect and the easiest way to generate transverse magnetization. You might think of these pulses as being infinitly short and with just the power to get the pulse angle you need. They actually can do the impossible, you can even do perfect spin specific pulses (impossible to do experimentally if two spins have overlapping transitions!).

There are lots of function in GAMMA that do ideal pulses (any angle, any phase, any selectifity). But **you MUST know** that these functions come in two flavors:

1 Those that operate directly on the spin system (density operator)

2 Those that produce pulse "propagators" that can be used repeatedly in a simulation.

If you just need a quick pulse in some simulation just a function that does a direct pulse on the system. If you are doing some long and involved multi-dimensional experiment simulations where the same pulse is repeately applied then use the latter, it will conserve both CPU time and memory use. If you don't know which to use don't bother, they both do the same thing if applied in your GAMMA program correctly.

Here's some code to demonstrate these things:

```
#include <gamma.h>
main()
  {
  spin_sys sys;                                  // System "sys"
  sys.read("test.sys");                          // System reads itself from file test.sys
  gen_op sigma0 = sigma_eq(sys);                 // Equilibrium density operator
  gen_op sigma1 = Iypuls(sys, sigma0, 90.0);     // Apply 90y ideal pulse (direct)
  gen_op U90y = Iypuls_U(sys,90.0);              // Pulse propagator for 90y
  sigma1 = evolve(sigma0, U90y);                 // Again the 90y pulse (with prop)
  sigma1 = Fx(sys);                              // About the same thing!
  gen_op sigma2 = Ixpuls(sys, sigma1, 33.3);     // Now pulse about x, angle 33.3 deg.
  sigma1 = Iypuls(sys, sigma0, 2, 18.9);         // Apply 18.9 deg y pulse to 3rd spin
  gen_op U90x1H = Ixpuls_U(sys, "1H", 90.0);     // Propagator for 90x on protons
  gen_op U90 = Ixypuls_U(sys, "51V", 45.0, 90.0);// 90 pulse, phase 45, on vanadiums
```

```
}
```

Enough? There's more. Note that I used "spin_sys" in the above program. That's because these pulse functions don't care about things like chemical shifts and coupling constants, that doesn't affect them at all. What would happen if you used "spin_system" instead in this program? NO EFFECT AT ALL. If you just want to see that a 90y pulse on FZ produces FX just use class spin_sys. If you want to generate a 1D NMR spectrum the use class spin_system because you'll need those shift and J values in other parts of your program. Class spin_system would work for watching FZ -> FZ but class spin_sys won't make it easy for you to make a 1D spectrum. Get it?

If, by some odd circumstance, you need to pulse with a different spin selectivity than what I've shown above there is indeed a way to do it. Say you want to pulse only spins 2, 3, & 6, what you do is set their spin flags in the spin system (just on/off switches that don't affect anything in particular) and call a special ideal pulse function that is active only on the spins who have their flags set. Have a look in the ideal pulse documentation for specifics, it's not a big stretch.

Remember, we are dealing exclusively with ideal pulses in this section. Other sections will cover square pulses, shaped pulses, pulse trains, and how to include relaxation effects during the pulses.

If all of this is stuff about density operators confuses you, get away from the quantum mechanics and look at the GAMMA treatment of the Bloch equations and magnetization vectors. You can do pulses and delays in that context too.

4 Plotting

4.1 Introduction

This chapter discusses methods of visualizing output from GAMMA simulations. Since each simulation in GAMMA is produced by a C++ program, the user always has the freedom to produce output to screen or file with the standard I/O functions available in C and C++ as well as the ability to link his/her program(s) to other I/O libraries. However, GAMMA has modules which interface to some of the more common plotting and manipulation programs.

4.2 Chapter Contents

4.3 Plotting Figures

4.4 Overview

GAMMA provides interfaces between itself and the formats supported by several useful software packages. This scheme is depicted in the following diagram.

GAMMA Supported I/O

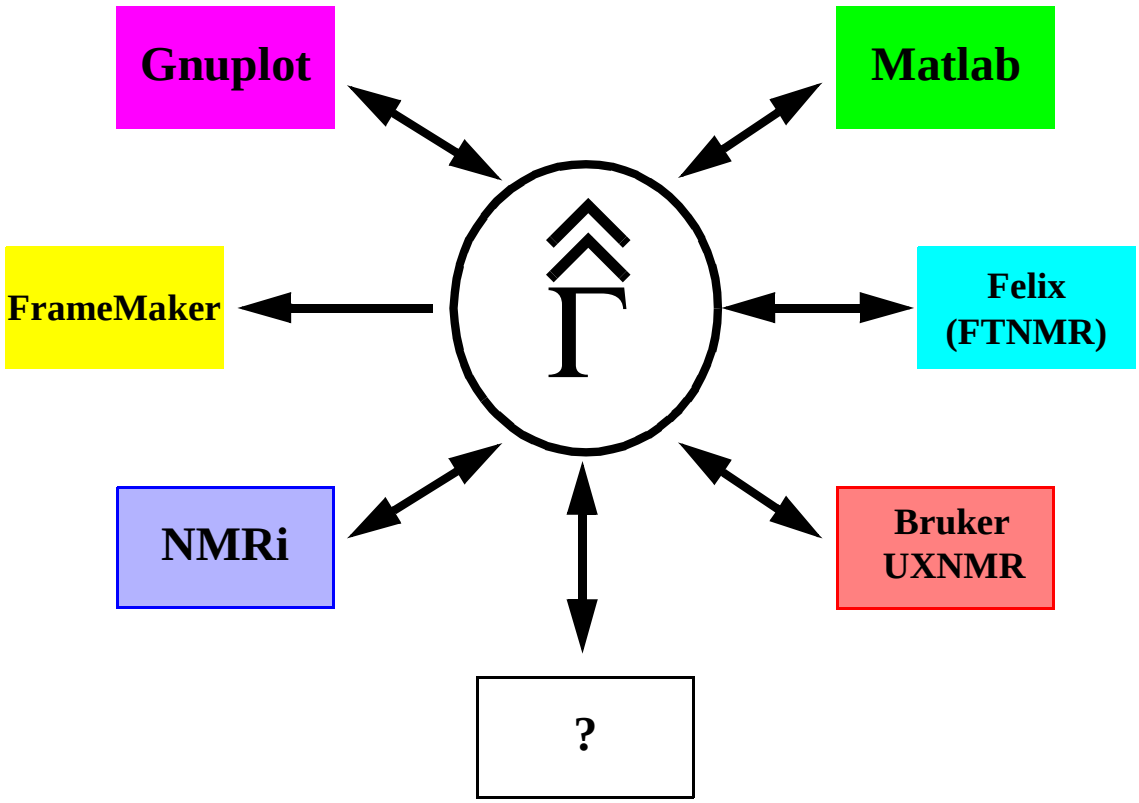


Figure 0-1 : Some of the programs with which GAMMA can easily interact. There are other prgrams that also have been used with GAMMA, those that come to mind at the moment are SigmaPlot, Deltagraph and XMGR. These take ASCII input and need no special interface.

GAMMA has the ability to read and write data in the formats used by the programs in the figure. Thus, GAMMA may be used to directly produce data in a format specific to any of these, or used to swap data between the different programs. This allows the user to take advantage of any data processing provided in these programs and any additional program which perform data conversions on them.

Keep in mind that GAMMA itself has only rudimentary abilities for graphical display and signal processing. It would be foolish to compete with professional and/or establish public domain programs that performs such tasks well. Furthermore, it is difficult to support all plotting and terminal devices, there are plently of excellent software packages on the market which already perform this duty. Our aim is simply to provide a means of reading and writing data files in the formats utilized

by such programs. A nice consequence of supporting multiple formats is that GAMMA may also be used to perform format conversions.

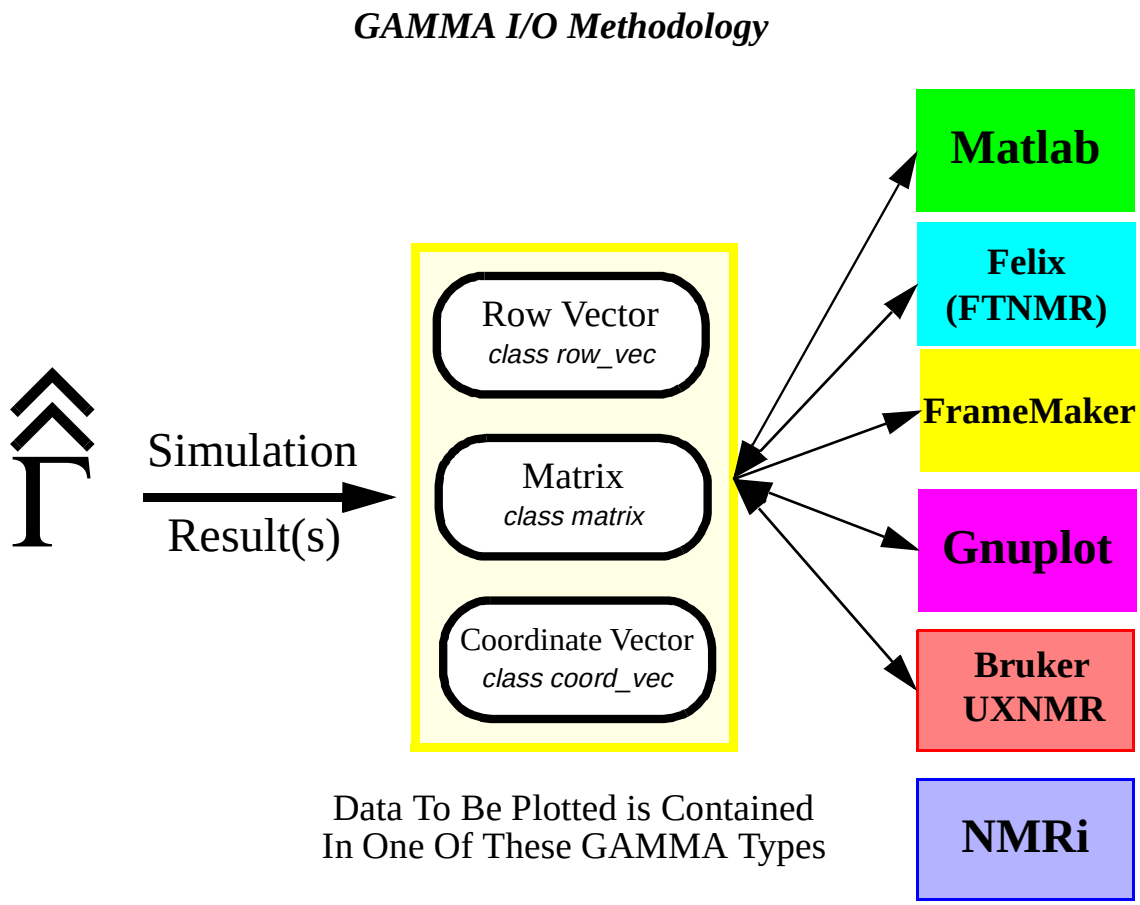


Figure 0-2 : Some of the programs with which GAMMA can easily interact.

We should mention that there are **currently NO direct plots from GAMMA to the screen, NOR direct output to any specialized printing and plotting devices.** There are many ways to do **"indirect" plots** so that your programs will **interactively display, print, and/or plot**. These are just GAMMA programs which output their data into one or more of the supported formats, then call the associated program(s) from within the GAMMA program, having it perform the visual and/or hardcopy output. In particular, see the Gnuplot section for programs which plot to screen interactively.

4.5 Gnuplot

4.5.1 Description

One way to have your plots appear on the screen during the course of a GAMMA simulation is to have your program output its data into a file that is compatible with the Gnuplot program and then call Gnuplot from within your program while it is running. Alternatively you can use Gnuplot to view results of GAMMA simulations after program completion.

Typically, a GAMMA program fills up a vector or matrix with simulated data. When the number crunching aspect is complete, one of the gnuplot functions provided in GAMMA is called with the vector or matrix as one of the function arguments. The gnuplot function writes out the data to an external file in a format which is readable with gnuplot (usually ASCII). If the plot is desired to be viewed during program execution then another call is made to the system, either using other GAMMA gnuplot functions or with explicit code. To view the plot using gnuplot after program completion, gnuplot is started and the appropriate commands issued to read the file created by the GAMMA program.

Keep in mind that you must have previously obtained and installed Gnuplot on your system. Instructions on doing should be included with the GAMMA installation documentation as well as on the GAMMA WWW site. If you need more details on how the GAMMA functions work have a look at the Graphics & I/O documentation as well as the GAMMA Gnuplot module source codes.

4.5.2 1D - Plots

The simplest type of plot is a 1D-plot created from a data vector with the function **GP_1D**. With this function the horizontal axis is the point index of vector and the vertical axis contains the data value (either reals or imaginaries).

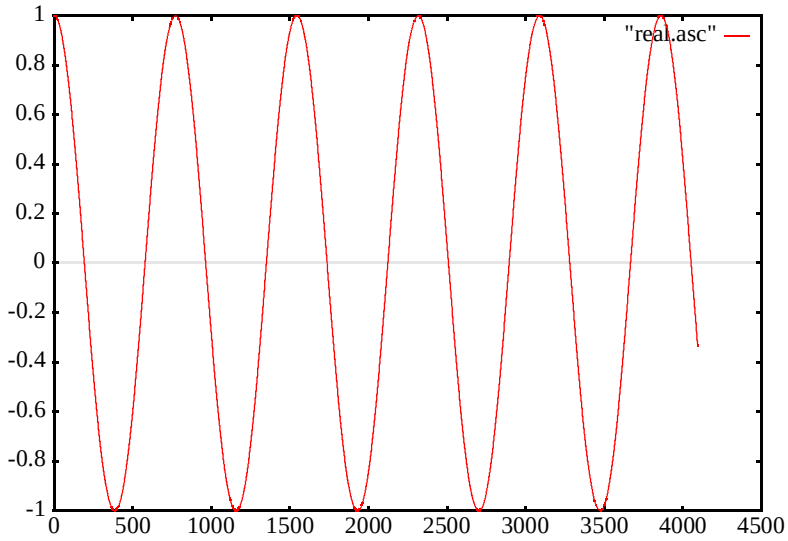
This program demonstrates how to have a 1D plot sent interactively to the screen using gnuplot. Note that in order for it to work, the system command “gnuplot” must be known to the user running the program.

```
#include <gamma.h>                          // Include GAMMA itself
main ()
 {
 int N = 4096;                              // We'll plot this many points
 row_vector data(N);                        // Here's a vector of points
 double rval, ival;                         // More temporaries
 for(int i=0; i<N; i++)                     // Now we'll fill up the vector
  {                                         // putting a cosine into the
  rval = cos(33.33*double(i)/double(N-1));  // real part
  data.put(rval, i);
  }
 GP_1D("real.asc", data, 0);                // Write real points to ASCII file
 GP_1Dplot("real.gnu", "real.asc");         // This will plot points in "real.asc"
 cout << "\n\n";                            // Keep the screen nice
 }
```

The call to the function **GP_1D** produces an ASCII file called “real.asc” that is usable by the Gnuplot program. It will contain the cosine function which was put into the row_vector **data**. The call to the function **GP_1Dplot** will make a plot on the screen of the data in **real.asc** during program

execution. It first makes an ASCII gnuplot “load file” called **real.gnu** then runs gnuplot using the commands in the load file.

The following figure is “roughly” the plot that appears on the screen when you run the program. What I’ve done here is re-run gnuplot after program completion, plotted the **real.asc** file, then output the plot into this document (using Gnuplots MIF output).



There are a couple of very nice Gnuplot features worth mentioning here. First and foremost is that it is a program in public domain. Not only does the user not have to pay for it, one has access to the entire source code and it runs on almost all common computer architectures. Second, Gnuplot has many different output formats. That means that you can get your figures in PostScript, MIF (as was done here) for FrameMaker, LaTex, PBM, even GIF. There is a Gnuplot website and it think it has its own newsgroup somewhere. There are many people who claim to use this software exclusively to produce publication quality plots.

4.5.3 Parametric Plots

Using a slight variation to the previous program one can make parametric plots which use both the reals and imaginaries of a GAMMA vector. For example,

```
#include <gamma.h>                          // Include GAMMA itself
main ()
 {
 int N = 4096;                              // We'll plot this many points
 row_vector data(N);                        // Here's a vector of points
 double rval, ival;                         // More temporaries
 for(int i=0; i<N; i++)                     // Now we'll fill up the vector
  {                                         // putting a cosine into the
  rval = cos(33.33*double(i)/double(N-1));  // real part
  data.put(rval, i);
  }
 GP_1D("real.asc", data, 0);                // Write real points to ASCII file
 GP_1Dplot("real.gnu", "real.asc");         // This will plot points in "real.asc"
 cout << "\n\n";                            // Keep the screen nice
 }
```

he simplest type of plot is a 1D-plot created from a data vector with the function **GP_1D**. With this function the horizontal axis is the point index of vector and the vertical axis contains the data value (either reals or imaginaries).

This program demonstrates how to have a

4.6 FrameMaker

4.6.1 Description

GAMMA provides functions for generation of figures suitable for direct use in FrameMaker (see http://www.frame.com). The GAMMA functions are handed data, typically a matrix or vector, and produce disk files in MIF (Maker Interchange Format) format or in the MML (Maker Mathematical Language) format. Plots or data structures are then seen by simply opening the file with FrameMaker. Such output may then be graphically manipulated and/or incorporated as part of a document and data such a matrices placed into FrameMaker equations. Plots produced in this manner can be printed on a laserprinter in PostScript, colorized to make transparencies, converted into HTML, etc.

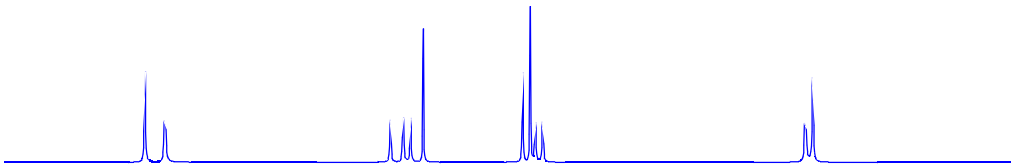
One important point to realize is that these FrameMaker files are not changeable into other formats with any GAMMA based code. If your data is valuable, and you wish to use it again in GAMMA, it should be stored to disk in one of the other formats so that GAMMA can retreive and manipulate it once again. There are some FrameMaker and public domain programs which can convert FrameMaker files to other formats, but they must be obtained independently from GAMMA.

To see all specifics regarding these functions look in the GAMMA FrameMaker Documentation.

4.6.2 1D Plots

The simplest type of plot is a 1D-plot created from a data vector with the function FM_1D. With-this function the horizontal axis is the point index of vector and the vertical axis contains the data value. An example would be the simulated NMR spectrum shown below.

Ninety Ideal Pulse On A 4 Spin Proton System.

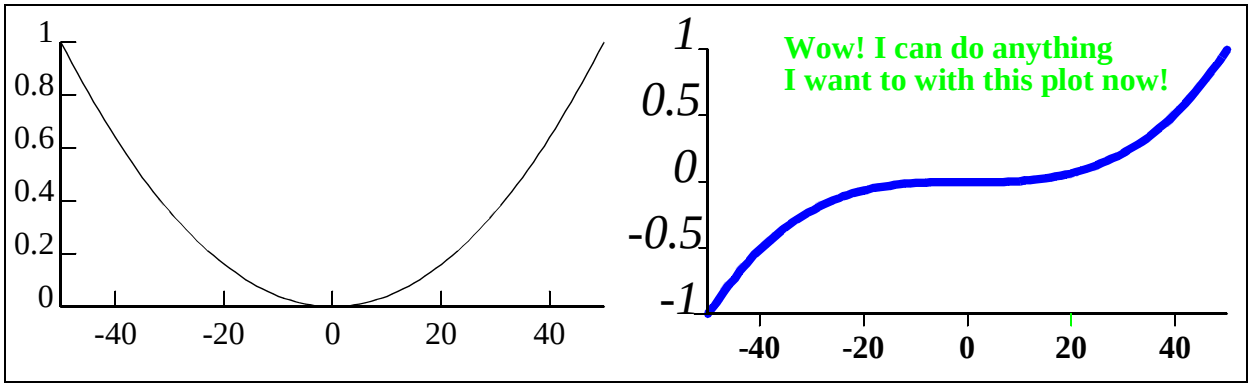


Have your GAMMA program fill up a row vector with the simulated data points you wish to plot. Then make a function call to FM_1D with the row vector and an output file name given as arguments. The output file can be read by FrameMaker. Please note that GAMMA takes no time to make “pretty” output, you can do the cosmetic work within FrameMaker!

Here is a simple example:

```
#include <gamma.h>
main ()
 {
 row_vector vx(101);                    // 1-dim. data block (or use row_vector)
 double x, y;
 for(int i=0; i<101; i++)               // Fill up data block
  {
  x = double(i-50);
  y = x*x*x/125000;                     // cubical parabolic in imaginaries
  x = x*x/2500;                         // parabolic into reals
  vx.put(complex(x,y),i);
  }
 FM_1D(“FM.mif”,vx,10,5, -50, 50, 1);   // output FM.mif with both plots
 }
```

When compiled and run it will produce a file called FM.mif. When that file is subsequently read by FrameMaker the following plots will appear. The one on the left has been left (except for resizing) as GAMMA produced, the one on the right has been cosmetically enhanced just to show what you can do to the plot in FrameMaker.



4.6.3 Multiple 1D - Plots

You can easily output several plots into the same graph by using the function FM_1Dm. Instead of providing a single vector to the function the user just provides an array of vectors.

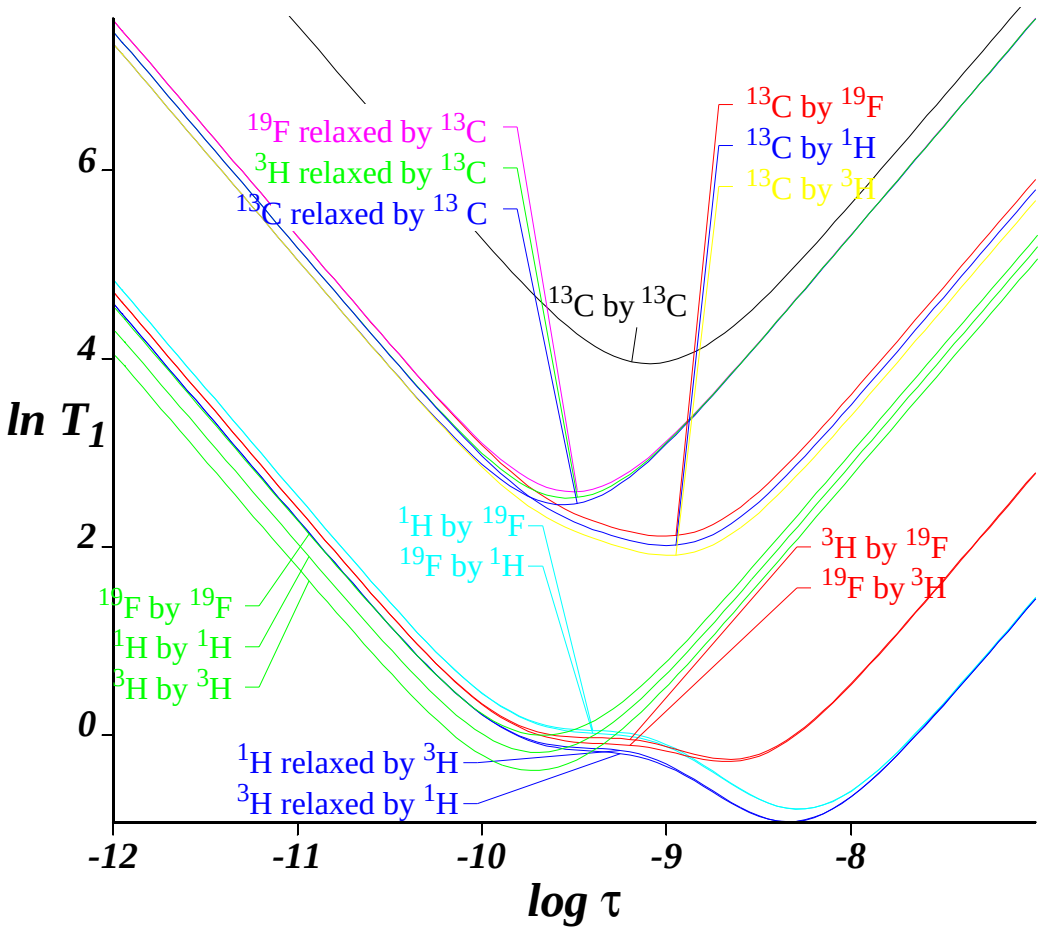


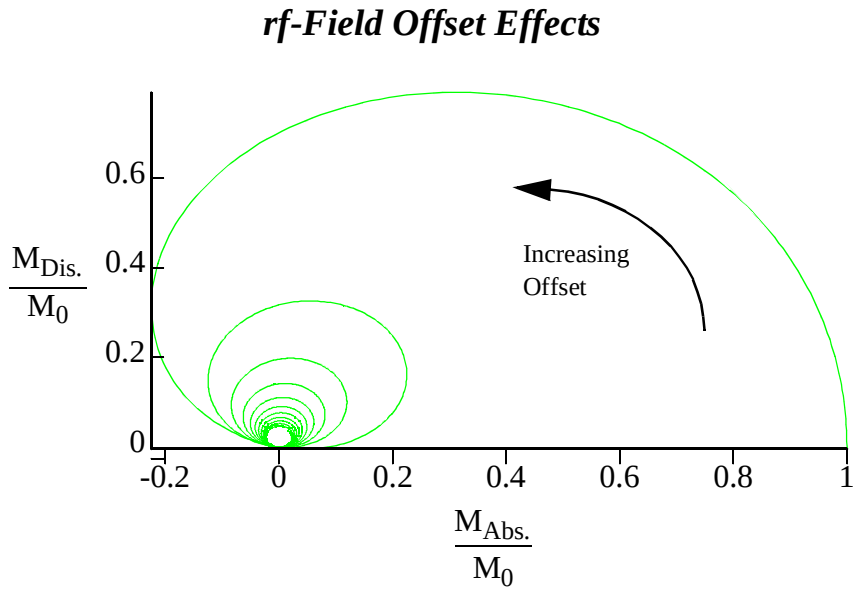
Figure 0-3 Dipolar longitudinal relaxation times for several spin pairs versus correlation time. The

distance between the spins was kept at 2A and the field strength set to 500 MHz.

The above plot was made with a simple GAMMA program using the FM_1Dm program (see GAMMA DIPOLAR RELAXATION DOCUMENTATION).

4.6.4 xy-Plane Plots

Plots in the xy-plane can be produced with the FrameMaker function FM_xyPlot. Unlike the function FM_1D, this function need not have the horizontal coordinate always increasing. The plot below is a typical example. It has been annoted and resized in FrameMaker after creatation with GAMMA.



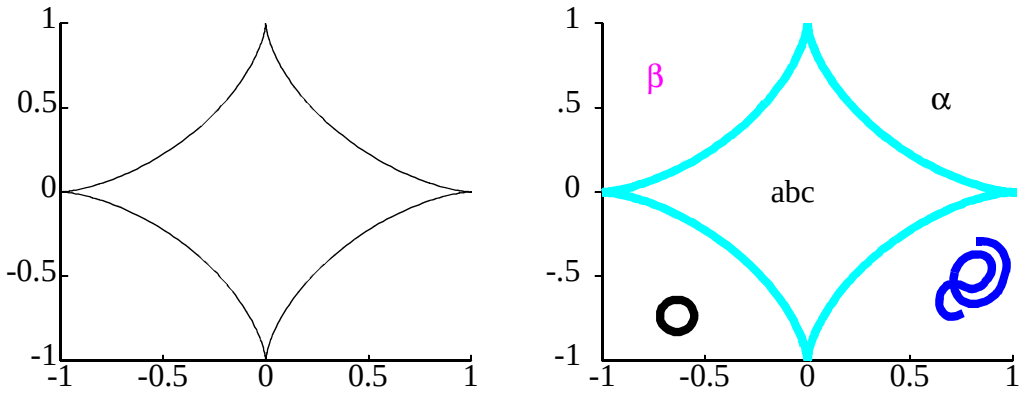
Have your GAMMA program fill up a row vector with the simulated data points you wish to plot. Then make a function call to FM_xyPlot with the row vector and an output file name given as arguments. The output file can be read by FrameMaker. Please note that GAMMA takes no time to make “pretty” output, you can do the cosmetic work within FrameMaker!.

Here is a simple example:

```
#include <gamma.h>
main ()
 {
 row_vector vx(360);                  // create a data block
 double x,y,theta;                    // declare needed variables
 for(int i=0; i<360; i++)             // loop through 360 degrees
   {                                  // fill up block with Astroid
   theta = i*2.0*PI/360.0;            // also called a Hypercycloid of four cusps
   x = cos(theta);                    // x = a*[cos(theta)]**3, here a = 1
   y = sin(theta);                    // y = a*[sin(theta)]**3, here a = 1
```

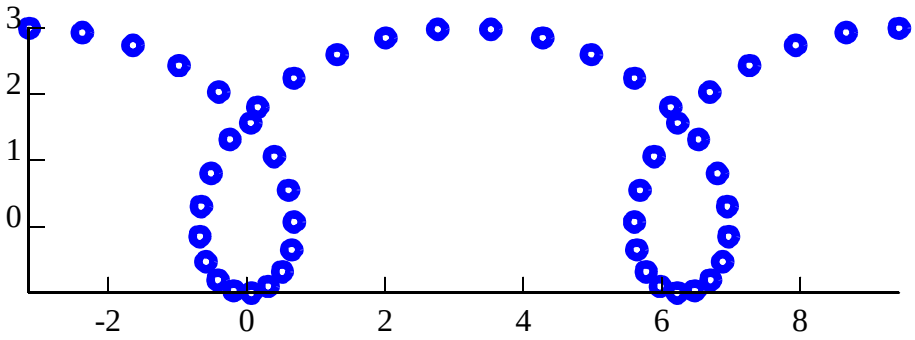
```
  x = x*x*x;
  y = y*y*y;
  vx.put(complex(x,y), (i);
  }
FM_xyPlot("astroid.mif", BLK);          // output FrameMaker .mif plot file
}
```

When compiled and run it will produce a file called astroid.mif. When that file is subsequently read by FrameMaker the following plot on the left will appear. The one on the right is just a copy that I've jerked with within FrameMaker.



4.6.5 Scatter Plots

Scatter plots can be generated for FrameMaker with the function FM_scatter. These are similar to plots produced with the function FM_xyPlot except that the plots are not connected and can be individually plotted with symbols or characters.



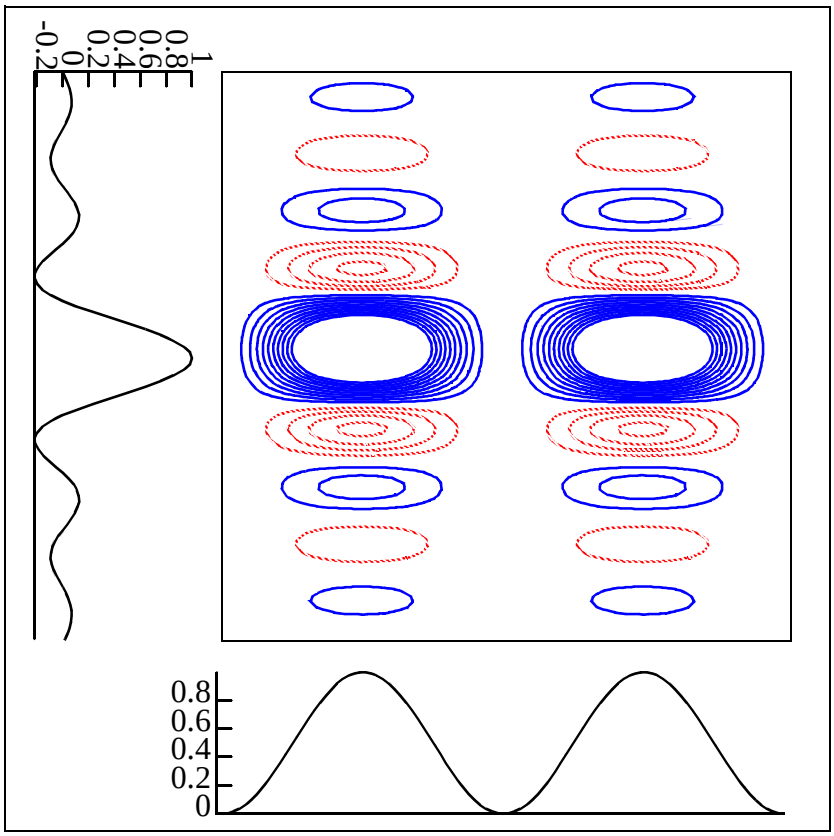
I forget why I even wrote this function now. Perhaps I used it to have my point plotted on top of a theoretical curve by blending a graphic like the one above with another one with a continuous line. Here's the code for the one above.

```
#include <gamma.h>
main ()
 {
 block_1D BLK(50);                          // create a data block
 double x,y,theta;                          // declare needed variables
 double a,b;                                // declare needed variables
 for(int i=0; i<50; i++)                    // loop through 50 points
   {                                        // fill up block with Prolate Cycloid
   a = 1;
   b = 2;
   theta = -PI + i*(4.0*PI/49.0);           // angles span -pi to 3pi
   x = a*theta - b*sin(theta);              // x = a(theta) - b*sin(theta), here a=1, b=2
   y = a - b*cos(theta);                    // y = a - b*cos(theta)
   BLK(i) = complex(x,y);
   }
 FM_scatter("FM.mif", BLK, 0, .1, 14, 5);   // output FrameMaker .mif plot file
 }
```

You can set the symbol type to use in the figure either in the function call or afterwards in FrameMaker..

4.6.6 2D Contour Plots

2D contour plots are produced for FrameMaker with function FM_contour. Each contour is an individual graphic object which can be manipulated. For example, the negative contours can selectively changed to dashed lines. Overall plot height & width can be set within FrameMaker.



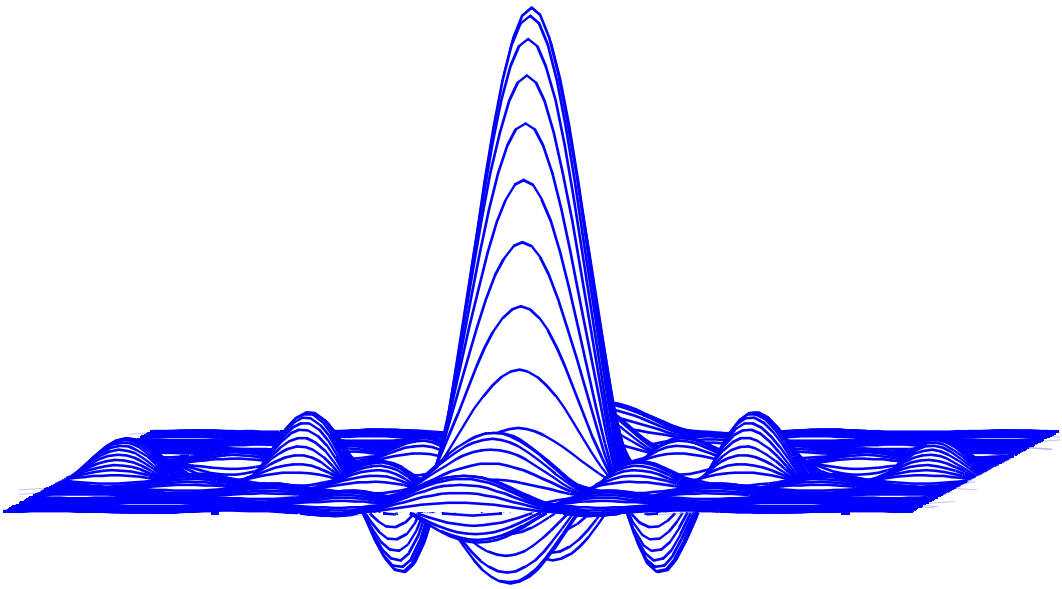
The example plot has be altered within FrameMaker and the 1D-plots added. Thsi program made it.

```
#include <gamma.h>
main()
 {
 matrix mx(101, 101);                    // create a 101x101 matrix for data
 row_vector BLK1(101), BLK2(101);        // create two 1D-data blocks of length 101
 BLK1 = sinc(101, 50, 10);               // use provided window sinc function
 BLK2 = sin_square(101, 50);             // use provided window sin squared function
 for(int i=0; i<101; i++)                // loop through and fill up the matrix
   for(int j=0; j<101; j++)
     mx(i,j) = BLK1(i) * BLK2(j);
 FM_contour("contour.mif",mx,.05,10,.05);   //create file FM contour file - contour.mif
 }
```

You can just use the computer to remove your T1 noise now! Read your spectrum (F1xF2) into a GAMMA matrix, output the contour plot into FrameMaker, Edit As You Wish (removing artifacts you don't want anyone to see, no more white out.....), Print to A Tranparency. Too Easy.

4.6.7 2D Stack Plots

Two dimensional stack plots are produced for FrameMaker with the function FM_stack. The function is called with a GAMMA matrix as an input argument.

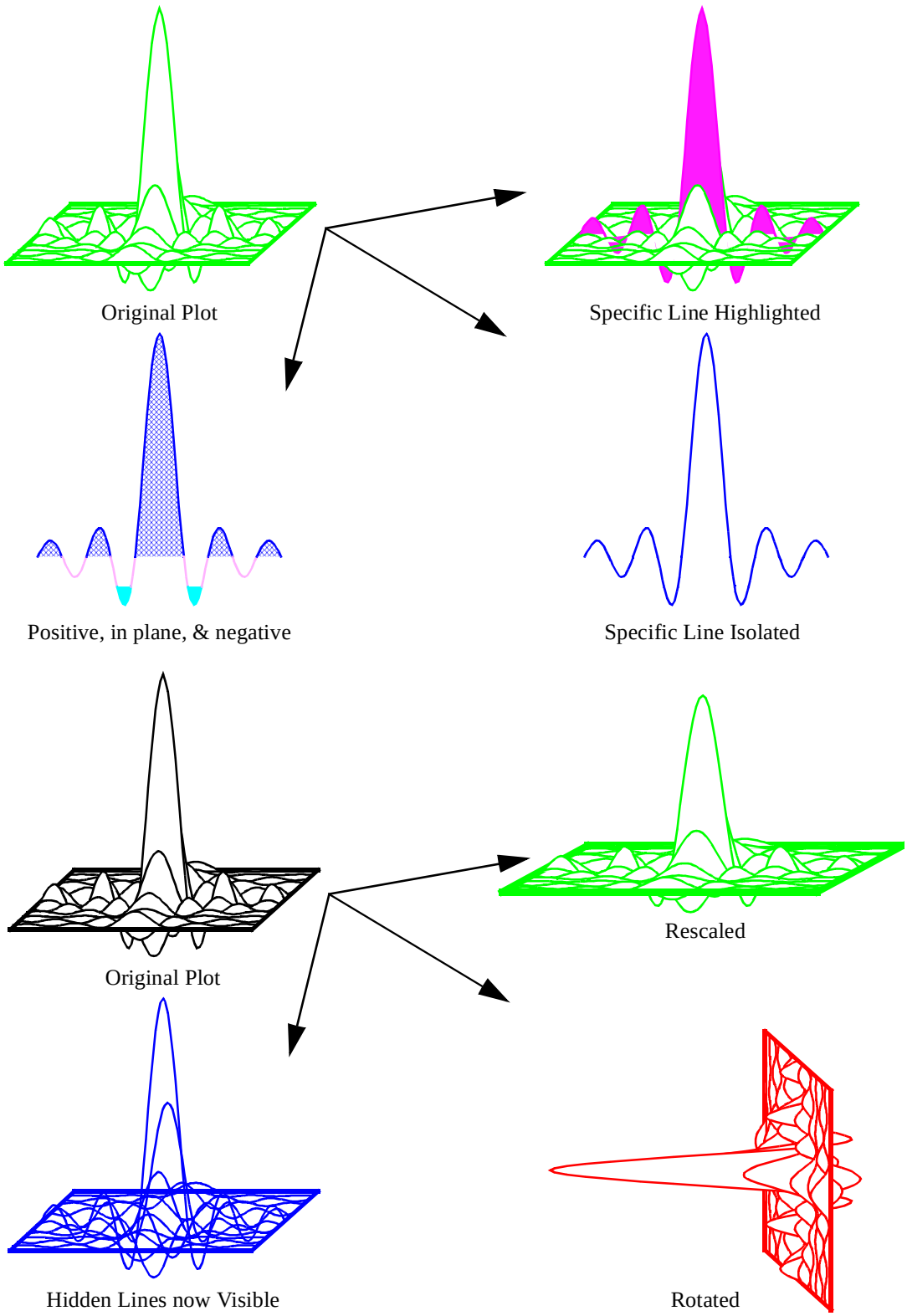


The function automatically used a hidden line algorithm (which can be removed in FrameMaker) and the user may set the skew. Each plotted row is an individual graphic object which can be manipulated. For example, a particular slice in the plot below was selectively shaded. The overall plot height and width can be altered as well within FrameMaker.

Here is a simple program, the one that produced the plot above.

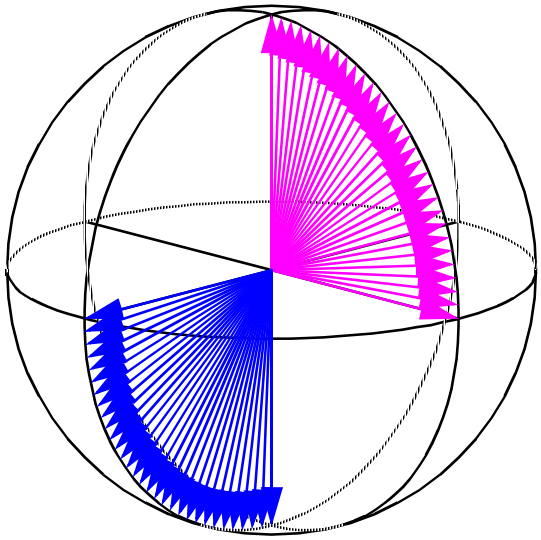
```
#include <gamma.h>
main()
 {
 matrix mx(101, 101);                    // create a 101x101 matrix for data
 block_1D vx(100);                       // create a 1D-data block of length 101
 vx = sinc(101, 50, 10);                 // use provided window sinc function
 for(int i=0; i<101; i++)                // loop through and fill up the matrix
   for(int j=0; j<101; j++)
     mx(i,j) = vx(i) * vx(j);
 FM_stack("stack.mif", mx, 0.02, 0.02, 1);   // output the FrameMaker .mif plot file
 }
```

The output file, stack.mif, can be read into FrameMaker and below are some of the manipulations that can be subsequently performed.


Original Plot
Specific Line Highlighted
Positive, in plane, & negative
Specific Line Isolated
Original Plot
Rescaled
Hidden Lines now Visible
Rotated

4.6.8 3D Sphere Plots

Three dimensional sphere plots are produced for FrameMaker with the function FM_sphere. This is real handy for making 3-dimensional plots of trajectories, for example

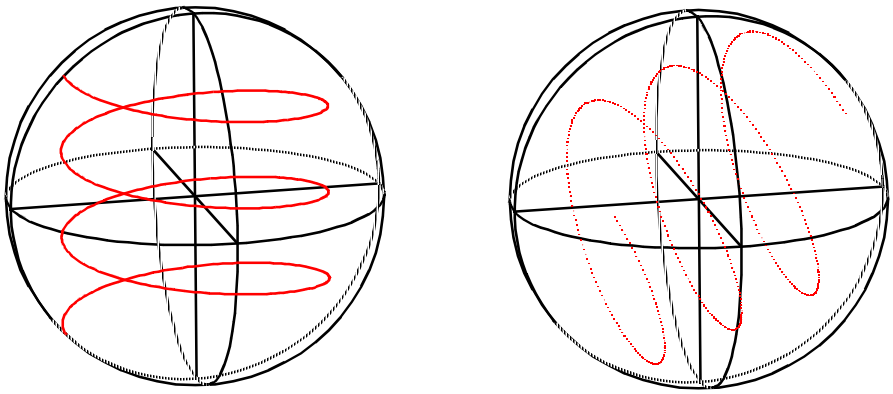


Unlike other FM functions, this function takes a vector of coordinates (class coord_vec) which are assumed to be in Cartesian space. There function allows the user to orient the sphere and to plot either with vectors from the origin to the points, with a line connecting points, or just with the points plotted individually. As with all other FM function, the user can fully manipulate the plot within FrameMaker after it is output by GAMMA. Here is a simple example program.

```
#include <gamma.h>
main ()
 {
 coord_vec data1(500);                  // Declare a 500 point coordinate vector
 coord_vec data2(500);                  // A second coordinate vector
 double xx, yy, zz;                     // Declare needed variables
 double theta;                          // Declare an angle variable
 for(int i=0; i<500; i++)               // Fill data1 with a spiral
   {
   theta = i*6.0*PI/499.0;
   xx = 3.0*cos(theta);
   yy = 3.0*sin(theta);
   zz = 3.0 - (6.*i/499.);
   data1.put(xx, yy, zz, i);
   }
 data2 = data1.rotate(90,90,0);         // Set 2nd coordinate vector to rotated data1
```

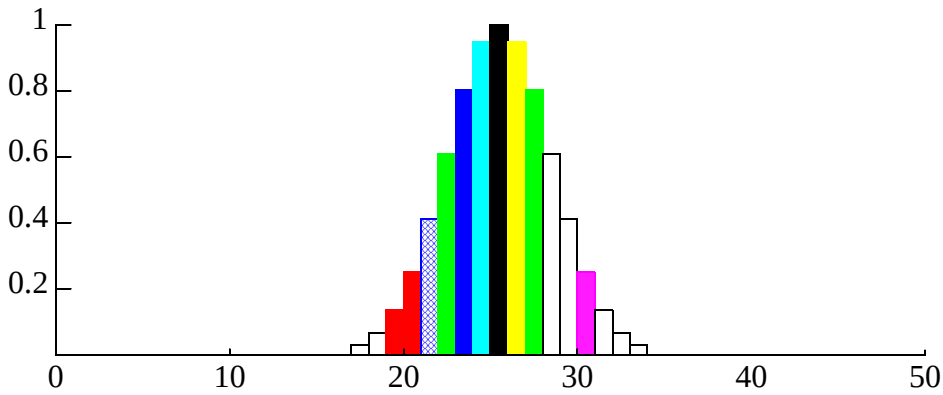
```
FM_sphere("FM_sph2a.mif", data1,0);      // Output FrameMaker file FM_sph2a.mif
FM_sphere("FM_sph2b.mif", data2,1);      // Output FrameMaker file FM_sph2b.mif
}
```

The program will produce two plot files, FM_sph2a.mif and FM_sp[h2b.mif which are shown below. I marked the points in red within FrameMaker after the fact.



4.6.9 Histograms

Histogram plots can be generated for FrameMaker with the function FM_histogram.



Here is a little program that illustrates how to make FrameMaker histograms in a GAMMA program.

```
#include <gamma.h>
main ()
 {
 row_vector vx(51);                      // create a data block
 row_vector vx1 = Gaussian(51, 25, 3);   // fill up data with Gaussian
```

```
for(int i=0; i<51; i++)
  vx(i) = complex(i, Re(vx1(i)));
FM_histogram("FM.mif", vx, bins);          // output FrameMaker .mif plot file
}
```

I colored some of the boxes in FrameMaker after reading the file FM.mif into that program. Perhaps one of the nicer uses of this function is to make cool plots of your pulse shape functions. Once again we see the typical scheme for getting plotted output from GAMMA. We do something to fill some array, in this case a row vector, with some simulated data we wish to plot. Then one (or more) of the plot functions is called with the array as one of the arguments.

4.6.10 Matrix Output

Matrices can be output for FrameMaker equations with the function FM_matrix. For example, the following double commutation superoperators given below were incorporated directly from GAMMA into this document.

$$[T^D_{2,1}(kl), [T^D_{2,-1}(kl), \quad]]$$

$$\frac{1}{16}\begin{bmatrix}
-2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -3 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & -3 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & -4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & -1 & -2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & -2 & 0 & 0 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -2
\end{bmatrix}$$

$$[T^D_{2,-1}(kl), [T^D_{2,1}(kl), \quad]]$$

$$\frac{1}{16}\begin{bmatrix}
-2 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & -1 & -2 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -3 & 0 & 0 & 0 & 0 & -1 & -1 & 0 \\
1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -2 & -1 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & -1 & -2 & 0 & 0 & 0 & 0 & 1 \\
0 & -1 & -1 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -4 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -3 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & -1 & -3 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & -2
\end{bmatrix}$$

Here is a simple example of how this is done.

```
#include <gamma.h>
main ()
{
 matrix mx(3, 4, complex0);          // Make a 3x4 array, filled with zero
 for(int i=0; i<mx.rows(); i++)      //Put 6's in the first column
   mx.put(6, i, 0);
 for(int j=0; j<mx.cols(); j++)      // Put i's in the first row
   mx.put(complexi, 0,j);
```

```
mx.put(complex(2,3), 1,2);          // Set <3|mx|4> to 2+3i
FM_Matrix("FM.mmf", mx);            // Output FrameMaker .mmf file
}
```

Here is the output from the program, the contents of file FM.mmf read by FrameMaker.

$$\begin{bmatrix} 1 \cdot i & 1 \cdot i & 1 \cdot i & 1 \cdot i \\ 6 & 0 & (2 + 3 \cdot i) & 0 \\ 6 & 0 & 0 & 0 \end{bmatrix}$$

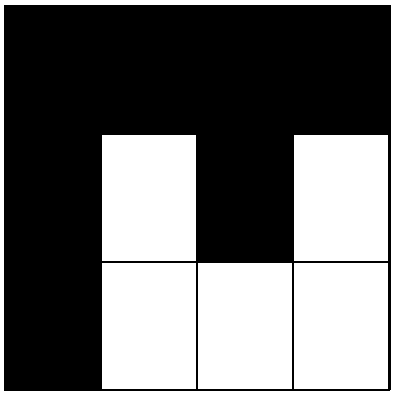
4.6.11 Matrix Plots

While the previous function is great for putting matrices into your documents (remember cout << mx will print any array to the screen), often the arrays in GAMMA programs are just too darn big to look at. Every once in a while we just need a graphical representation of which elements are non-zero rather than what the actual elements are. For that purpose you can use the function FM_Mat_Plot.

For an example, we'll just reuse the previous program but with the FM_Mat_Plot function instead of FM_Matrix.

```
#include <gamma.h>
main ()
 {
 matrix mx(3, 4, complex0);          // Make a 3x4 array, filled with zero
 for(int i=0; i<mx.rows(); i++)      //Put 6's in the first column
   mx.put(6, i, 0);
 for(int j=0; j<mx.cols(); j++)      // Put i's in the first row
   mx.put(complexi, 0,j);
 mx.put(complex(2,3), 1,2);          // Set <3|mx|4> to 2+3i
 FM_Mat_Plot("FM.mif", mx);          // Output FrameMaker .mmf file
 }
```

Here is the output from the program, the contents of file FM.mif read by FrameMaker.



Those familiar with FrameMaker should not that the result of this function is a graphical object whereas the result of the previous function goes into an equation.

4.7 Felix

4.7.1 Description

GAMMA has some knowledge of the Felix NMR processing program. It contains a class for performing I/O to and from Felix files (and to the older FTNMR).