

[home](#)

[-README](#)

[-INSTALL](#)

[-Windows.README](#)

[-Readme.BLAS](#)

[-Speed Tests](#)

[-Compilation, blcc](#)

Level 0::utils/IO

[-Matlab](#)

[-Utils/Constants](#)

[-Timer](#)

[-Random](#)

[-VNMR](#)

[-Parameters](#)

[-Parser](#)

[-ScriptParse](#)

[-XWINNMR](#)

[-SpinSight](#)

[-wavestream](#)

Level 1::containers

[-Coordinates](#)

[-Range/Spread](#)

[-Vector](#)

[-matrix](#)

[-grids](#)

Integrated Libs

[-MPI](#)

[-MINUIT](#)

Level 2::Function

Structs

[-ODE solvers](#)

[-Shapes](#)

[-TimeTrain](#)

[-Stencils-Vector](#)

[-Stencil-Grids](#)

[-Integrate](#)

Bloch Equations

[-Isotope](#)

[-Bloch Parameters](#)

[-ListBlochParams](#)

[-Gradients](#)

[-RotatingGrid](#)

[-Pulses](#)

[-BiotCoil](#)

[-Biot](#)

What Is It?

- An expression template library of generic algorithms to perform large scale spin simulations...HOWEVER...all the classes up before 'Isotope' on the side menu are completely generic and form a set of fast data structures....so you can use them in other apps. The classes after Isotope are a specific implementation and usage of the base classes (but are sufficiently complex to warrant their own inclusion here)
- **The library's Goal:**
 - speed
 - ease of use for many aspects of scientific programming in an object-oriented framework
 - apply modern algorithmic and programming formalism (Objects, expression templates, iterators, etc)
 - incorporate existing 'proven' numerical techniques and parallel structures ([STL](#), [MINUIT](#), [MPI](#), [FFTW](#), etc.)
 - to make simulations of spins in unconventional situations possible in real time.
 - make the documentation a learning tool and a simple, useful interface to the code.
 - provide a vast array of examples to aid in program creation and understanding of spin physics
- This library is in a CONSTANT state of change, updates, improvements, etc. The documentation is typically one step BEHIND the current code status as something may change, or has been well tested...The below 1.0 version releases are not complete releases, but beta snapshots..with each new increment closer to achieving the final goals stated above.
- There are now MANY of examples, ranging from simple class demonstrations (like "this is how to use the matlab IO class") to complex simulation packages ("solid" an application to rival Simpson in terms of speed and ease of use), to single focus programs (the Bloch simulations investigating certain interactions and other novel ideas)

Updates and such

- **Since 8.22.02 ---(10.18.02)**
 - Several Improvements and additions (this should technically be called BlochLib-1.1, but as I am the version control, I care not to alter every little thing to 1.1)
 - 3 new data readers/writers...
the XWINNMR--> reads the parameters and FIDs from an XWINNMR directory structure (both 1 and 2Ds),
SpinSight--> reads the parameters and FIDs from a SpinSight directory structure (both 1 and 2Ds),
wavestream --> reads and writes .WAV audio files from many data types.
 - The Full matrix--matrix multiplication C*=A has been improved by ~30% IT CALCULATES A DYSON TIME ORDERED product...i.e. C=A*C (not the usual C=C*A) but

- MultiBiot
- Interactions
- Offset
- Relax
- BulkSus
- RadDamp
- DipoleDipole
- DemagFied
- Bloch
- BlochSolver

Quantum Mechanics

- SpinSys
- SpinOperators
- Spin/Space Tensors
- Rotations
- Csa
- Quadrupole
- Dipole-Dipole
- J-coupling
- HamiltonianGen
- SolidSys
- powder
- compute
- oneFID

All Examples

is ~30% (depending on the matrix size) faster then the C=A*C.

- the Complex Numbers now can have float or double sub precision (i.e Complex<float> and Complex<double>) the old 'complex' is still valid and is only a typedef of Complex
- The matrix has a new 'reshape' function that functions exactly as the matlab version (e.g. takes a 2x5 matrix to 1x10, etc)
- a faster 'copy' and 'assignment' (operator=) for the matrix of the SAME TYPE and STRUCTURE (uses the 'memcpy' function which is quite speedy)

- **BlochLib 1.0 ---- yippie.... 8.22.02**

- Several new things have been updated/fixed since the 0.9 version (as the list below can attest to)
- 1) NAMESPACES
 - The advent of making everything gcc 3.1 compatible required me to do some place the library into a namespace, thus in every source file you use "blochlib.h", you will need to include "using namespace BlochLib;" in the file as well.
- 2) should compile on both gcc 2.95 and gcc 3.1 (DO NOT USE GCC 3.0 it is WAY too slow)
- 3) ModulatedDemagField class fixed.
- 4) Documentation (over 1000 pages) should be complete aside from typos (which i am sure there are plenty).

- **update 8.2.02**

- The documentation drudges along...many a minor bug fixed...
- 1) fixed a minor problem with the rotations class when the rotor speed was set to 0 and when the rotor angle was not set to 0 (which thus effected some of the hamiltonians)
- 2) several enhancements and fixes to the matrix class involving resizing and other 'alteration' functions (getting rows, columns, etc) have been repaired. Under odd conditions they would not perform correctly, and this is now fixed.
- 3) the Relaxation Bloch Interaction has seen many improvement including the ability to relax off the (0,0,1) axis
- 4) The VNMR class has a 'ProcPar' class reader and the ability to open a 'directory' (the standard place where VNMR puts the procpar and the fid).
- 5) exceptions are now implemented over the old 'exit()' routines so you too can try and catch things.
- 6) a math 'Parser' class has been added which will parse an input string with the usual mathematical functions (like 2*(5+2)^(2)) that allows for global and local variable...the current implementation does not allow for complex numbers or multi input functions (you cannot do something like 'moo(9,0)' only 'moo(9)' will work).
- 7) a basic script interpreter ('ScriptParse') that parses variables assignments (A=B), loops and if/elseif/else statements. It was created with the ability to let you create your own scripting interface

by simply creating a new class with function that decides (called 'decide') what to do if it hits one of your own functions....

- 8) with those above new classes and the various bug fixes throughout the library, the Solid example/program has been DRAMATICALLY altered (current version...solid-2.0). It does away with the 'scripto' interface and incorporates its own mild scripting capabilities with dramatically altered syntax. (solid-1.6 and solid-2.0 files ARE NOT compatible at all). Currently it is a stripped down version of the old program (solid-1.6). The scripting interface now allows for dynamic sections, and arbitrary 2D spectra of any type...you can simply change a parameter and recollect the spectrum and it will be added to a 2D master list which will be saved as a matlab file. Mutli Spin systems can also be used so you can change the entire interaction set at any time (this usually needs to be performed for liquid-crystal DAS experiments). Point-To-Point 3D experiments are now possible (these really create a 2D data set). The same capability of pseudo phase cycling is still present. The input syntax is so much simpler and useful that it is a pity these things were not done sooner. It does not, however, perform the 'hamil' or 'fit' types of abilities the old version had before. BUT there is a stable 'hamil' program in the examples anyway which performs better anyway, and the fitting was so unreliable in the first place, it really did not need to be there. The 'bspec', 'opdet', and 'sym' options are no longer needed, as you can simply use the basic scripting interface to perform these anyway. The 'jump' type has also removed because of its complete lack of generality (thus making it not very useful)

- **update 6.30.02**

- The documentation is now databased! soon you will find search functions, and even other formats other then HTML

- **update 6.15.02**

- the ODE solvers have been updated with a better step size control (i.e. less silly errors in integrations)
- an another ODE integrator "stiffbs" has been created to handle STIFF equations...dare not try the "bs" or "ckrk" methods for stiff eqs as you will be sitting for a LONG time and the answers you will get will be full of errors...the "stiffbs" requires the Jacobian of your systems
- several general improvements and minor fixes to the Matrix class

- **since version 0.9 (6.10.02)... slow this is, but found a few bugs and added a few enhancements**

- the 'Parameters' has had a bunch of bugs removed..mostly dealing with multiple sections...the new functions of 'setParam' have also been added, as well as a few speed improvements in the class
- there is a 'Magnetic Coil Generator Function Registration' to allow you to write your own coil functions, and register them so that the 'MultiBiot' class can read them from your parameter files (see examples/blochsims/magfields/exsitu/Dcoil.cc for an example)

- RotatingGrids have been implemented (the classical analog of a spinning MAS experiment)...silly, i could rotate magnetics fields, but forgot about the spacial part...that should be fixed now (see examples/blochsims/mas/ for an example)
- the examples should be plenty documented and fairly complete in terms of rigor
- the 'StencilPrep' calculation has been parallelized...NOTE:: it actually can be slower in parallel with slow network connections.
- ListBlochParams now has a better implementation of the 'calcInitialConditions()' to allow you to loop through 'random' configs...

- **version 0.9...**The end is near...

- ALL the main data drivers, containers, and functionality has been included and implemented...now it is bug testing time and finalization of 'loose ends'
- What remains...
 - Documentation...hopefully as all the code design is done, the documentation can be written..
 - Diffusion? i am still debating to include diffusion in the Bloch_Interactions...it may be a bit too difficult, and slow for the current version of the lib...a thing to keep in mind for version 2 perhaps as the grid class would need to be rewritten.
 - Examples need to be documented better (the code need some comments as well as updating from the older version of the lib)
 - General bug fishing and testing
- updates:
 - another speed upgrade for the 'prop(matrix, matrix)' function improved by ~30%
 - a full MPI controller class (instead of the C functions, we have an object now) with both BLOCKING and NON-BLOCKING commands
 - the 'biot' class has been parallelized for optimum performance
 - the 'solid' program (included in the examples) has been heavily updated and optimized for parallel running as well a ton of other speed enhancements and minor bug fixes.

- **version 0.8...** SPEED improved...

---Now allows the [Fastest Fourier Transform in the West \(FFTW\)](#) library to be used as the main driver for Fourier Transforms of Vectors and Matrices...A very fast implementations for FFTs...i highly recommend using this library...simply type "--with-fftw=" in the configure path, and everything will be properly set.

---the matrix Multiply code (for full matrices) has been DRASTICALLY altered. There are currently 2 setups possible...as we all know, the machine

specific BLAS as ALWAYS much faster then anything a simple one man show can create, sooo i tries to create some 'okay' optimize code (meaning that if you do not have a machine specific BLAS package, everything still works, and with okay speed...ie.. it is still faster then the basic netlib matrix multiply and many other packages, but compared to what the vendors can give you , it will be ~2-3 times slower)...so if you want to use a vendor BLAS package, you will need to 2 things

- a) use the '--with-blas=' option in the configure
- b) Edit the top of 'container/matrix/matmatmul.h" to reference the correct function within the lib
- **INTEL (LINUX)::***just so you know, the current set up in 'matmatmul.h' is for Intel's MKL library and ATLAS libs.**

NOTE-->I have found that the [ATLAS](#) tuned libraries can be FASTER than the MKL libs on some Intel machines, i invite you to try testing them versus each other. Read the README.BLAS for more info on how to configure and use ATLAS for this library.

<http://developer.intel.com/software/products/mkl>

specific for Pentiums (and it is quite speedy, in fact i think it is a fast as can one can get)

- DO NOT use the 'lapack' lib or any Shared libs (i.e. no *.so) here, use either the 'libmkl_p3.a' (for pentium 3's) or 'libmkl_def.a' for everything else
- It requires the "pthread lib" on linux so do "--enable-pthread" in the configure line.....
- you will need to set "**OMP_NUM_THREADS**" in your environment if you have an SMP machine (careful, not setting this can cause mass havok if you also use MPI)
- **INTEL (CYGWIN/PC)::**Read the README.BLAS on how to make the best BLAS lib and incorporate it into the library. You will need 1) [cygwin](#), 2) [ATLAS](#)...please read my directions inside the windows readme file when configuring ATLAS...
- **SGI::** go here for the lib optimized lib
<http://www.sgi.com/software/scsi.html> (this has not yet been implemented in the library yet)
- **SUN::** go here for the lib optimized lib
http://docs.sun.com/htmlcoll/coll.118.3/iso-8859-1/PERFLIBUG/plug_bookTOC.html (this has not yet been implemented in the library yet, but SUN's library should come with the OS)

***if anybody sets this little file up for SUNs or SGIs (or anything else really) and gets it to work for those specific BLAS, then i will gladly include it in the config script (to differentiate between the platforms)...i do not have these machines to do this test myself....

- 1a) Lots of other speed improvements have been made throughout

the code making the Quantum Mechanical type simulation go ~5-10 times faster....(nice eh?)

- 2) The rotation class has had 'Cartesian' or 'spherical' or 'both' types of rotations are handled in an optimized way
- 3) a bug fix for the second order Quadrupole interaction

- **Version 0.7...**The beginning of the parallel world...some VERY BASIC [MPI](#) functions to allow one to compile code with or without the MPI library present...

- In the works...simple "get" and "put" methods for each container type (matrix, vectors, coords, etc.) are being written...
- NOTE:: this MPI implementation is not really designed for massively parallel systems (i.e. systems of 128 processors) but more the 2-10 processors range where distributing loops is advantageous, the normal container operations (things like "vector + vector") will not be parallel unless you create a specific function to do so (which should be simple once the main interface is complete)...
- look to the 'MPI' section of the documentation for more details and examples, and look to the 'INSTALL' section for the new configure options for MPI
- NEW:: a VNMR class called 'VNMRstream' that will read and write the binary formats (both 1D and 2D) from the VNMR (Varian) spectrometer system
- The general solid-state simulation application 'Solid' is now included as a glorified example in the 'examples' directory
- The very nice minimization package "MINUIT" from has now been included in the library using a heavily debugged and reworked '[C-MINUIT](#)' to allow inclusion into the shared lib as well as C++ apps

- **Version 0.6...**Ground Planned altered to encompass a more general approach to high field, low field, density and Particle pictures of the simulations or Objects within the simulation

- The old techniques will still work from version 0.5...but now you can/should specify 'What kind' of simulation one is doing "HighField, Density" or HighField Particle" etc.
- CHANGED:: 'Offset' and 'Relax' have been added as new Interactions as they have been taken OUT from the BlochParams
- NEW Magnetic field calculator from Coil geometry's over the 'XYZshape' object (class 'Biot')
 - TODO:: Some symmetry enhancements (use the symmetry of the coil if there is one to make the calculation faster..this is now in it INFANCY stage...(so carfeul using it, it can give you some weird results)
- NEW interactions::Offset, Relax,
 - Offset using Bfields calculated from Magnetic Field calculator
 - Dimensionless Dipole -->Quantum Type specification of coupling (i.e. "200 Hz")

- **Version 0.5...**the basic Quantum Mechanical framework to perform most Solid State simulations

- AS with the rest of this 'document-in-progess' some of the basic documents have been written..several others need to be written...i am purposely leave them out as the 'newer features' are not promised to remain as they are at this point...certain issues may arise later that require them to be retailored...but you can certainly look at the examples and source to get an idea of what they do....
 - 'non-linked' documents are those that have been written and function...my documentation skills lack the gusto..to write them with any speed....
- **Version 0.4** has been completed...as of yet still not nearly any final release, but it should be quite stable...
 - Updates from version 0.3
 - A more generic Shape-Grid Generator (including the ability to use multiple shapes to create a 'multi-faceted' grid array)
 - More XYZshape Functional generators
 - Temperature, Inhomogeneity, and Concentration(Moles) based on grid position (and those multifaceted shapes)
 - Large Speed improvement to the 'coord' class AND given it the ability to act as a very fast 'Fixed Sized Vector' for arbitrary sizes (it can be any size now...not just of length 3)
 - Stencils...the basic algorithms to calculate numerical derivatives and differential operations on data...ONLY the VECTOR implementations have been completed
 - Lyapunov Exponent calculator class for generic 'Jacobian' functions and input data.
It has been integrated into the 'BlochSolver' so that you may calculate Lyapunov exponents for your simulations FOR EVERY item...
 - Dipole-Dipole interaction (the finite-element approximation to the Spin-Spin Dipole interaction in High fields...)
 - Several improved Output functions for each class...very handy for debugging code and for visualizing the complex grid patterns/gradient things...
 - What remains to be done:
 - Stencil operations for Matrix and The NON-uniform grid arrays (as of now there are no above 2D array data structures) only Fixed sized Vectors, Dynamic Vectors, Matrix, Grids, and Parameters.
 - Diffusion...once the stencil operations over the 'non-uniform/non-constant' grid shapes are completed this will be done
 - Input...this is VERY weak...and without writing specific inputs for each class as of now, not existent (except for the BlochParams and ListBlochParams)...this is a secondary concern after the numerics have been completed

The Distribution README

BlochLib README

author:: Bo Blanton
last modified:: August 20 2002

THE MASSIVE DOCUMENTATION IS in HTML
go to {source-root}/doc/html/index.html

AND PDF
go to {source-root}/doc/pdf/
this file is >1000 pages, so careful with printing...

for the most current version of the documentation go to
<http://waugh.cchem.berkeley.edu/blochlib/>

you must have a frame enabled browser
(there is too much info to place on one page)

or if you are useing an older browser, or have
your font size set up to be much larger then
'normal', please go here
<http://waugh.cchem.berkeley.edu/blochlib/single.php>

The Most important Doc so far is the 'Matrix' it has some peculiar rules
and implementations...so READ IT

->For installation options see "INSTALL"

--version-1.0..the final source

---Some documentation still needs some fine tuning, but the source is done

---updates::

1) NAMESPACES

The advent of making everything gcc 3.1 compatable required
me to do some place the library into a namespace, thus in every
source file you use "blochlib.h", you will need to include
"using namespace BlochLib;" in the file as well.

2) should compile on both gcc 2.95 and gcc 3.1

(DO NOT USE GCC 3.0 it is WAY too slow)

3) ModulatedDemagField class fixed.

4) fixed a minor problem with the rotations class when the rotor
speed was set to 0 and when the rotor angle was not set to 0
(which thus effected some of the hamiltonians).

5) several enhancements and fixes to the matrix class involving
resizing and other 'alteration' functions (getting rows,
columns, etc) have been repaired. Under odd conditions they
would not perform correctly, and this is now fixed.

6) the Relaxation Bloch Interaction has seen many improvement
including the ability to relax off the (0,0,1) axis.

7) The VNMR class has a 'ProcPar' class reader and the ability to open

a 'directory' (the standard place where VNMR puts the procpar and the fid).

8) exceptions are now implemented over the old 'exit()' routines so you too can try and catch things.

when configuring use '--enable-exceptions' to allow them (they take longer to compile, the default is NO exceptions.

It raises a SIGINT as the default.

9) a math 'Parser' class has been added which will parse an input string with the usual mathematical functions (like $2*(5+2)^{(2)}$) that allows for global and local variable...the current implementation does not allow for complex numbers or multi input functions (you cannot do something like 'moo(9,0)' only 'moo(9)' will work.

10) a basic script interpreter ('ScriptParse') that parses variables assignments (A=B), loops and if/elseif/else statements. It was created with the ability to let you create your own scripting interface by simply creating a new class with function that decides (called 'decide') what to do if it hits one of your own functions....

11) with those above new classes and the various bug fixes throughout the library, the Solid example/program has been DRAMATICALLY altered (current version...solid-2.0). It does away with the 'scripto' interface and incorporates its own mild scripting capabilities with dramatically altered syntax. (solid-1.6 and solid-2.0 files ARE NOT compatible at all). Currently it is a stripped down version of the old program (solid-1.6). The scripting interface now allows for dynamic sections, and arbitrary 2D spectra of any type...you can simply change a parameter and recollect the spectrum and it will be added to a 2D master list which will be saved as a matlab file. Multi Spin systems can also be used so you can change the entire interaction set at any time (this usually needs to be performed for liquid-crystal DAS experiments). Point-To-Point 3D experiments are now possible (these really create a 2D data set). The same capability of pseudo phase cycling is still present. The input syntax is so much simpler and useful that it is a pity these things were not done sooner. It does not, however, perform the 'hamil' or 'fit' types of abilities the old version had before. BUT there is a stable 'hamil' program in the examples anyway which performs better anyway, and the fitting was so unreliable in the first place, it really did not need to be there. The 'bspec', 'opdet', and 'sym' options are no longer needed, as you can simply use the basic scripting interface to perform these anyway. The 'jump' type has also removed because of its complete lack of generality (thus making it not very useful)

12) the ODE solvers have been updated with a better step size control (i.e. less silly errors in integrations)

13) an another ODE integrator "stiffbs" has been created to handle STIFF equations...dare not try the "bs" or "ckrk" methods for stiff eqs as you will be sitting for a LONG time and the answers you will get will be full of errors...the "stiffbs" requires the Jacobian of your systems.

14) the 'Parameters' has had a bunch of bugs removed..mostly dealing

with multiple sections...the new functions of 'setParam' have also been added, as well as a few speed improvements in the class.

15) there is a 'Magnetic Coil Generator Function Registration' to allow you to write your own coil functions, and register them so that the 'MultiBiot' class can read them from your parameter files

(see examples/blochsims/magfields/exsitu/Dcoil.cc for an example)

16) RotatingGrids have been implemented (the classical analog of a spinning MAS experiment)...silly, i could rotate magnetics fields, but forgot about the spacial part...that should be fixed now

(see examples/blochsims/mas/ for an example)

17) the 'StencilPrep' calculation has been parallelized...NOTE:: it actually can be slower in parallel with slow network connections.

18) ListBlochParams now has a better implementation of the 'calcInitialConditions()' to allow you to loop through 'random' configs...

--version 0.9...The end is near...

--- ALL the main data drivers, containers, and functionality has been included and implemented...now it is bug testing time and finalization of 'loose ends'

---What remains...

1) Documentation...hopefully as all the code design is done, the documentation can be written..

2) Diffusion? i am still debating to include diffusion in the Bloch_Interactions...it may be a bit too difficult, and slow for the current version of the lib...a thing to keep in mind for version 2 perhaps as the grid class would need to be rewritten.

3) Examples need to be documented better
(the code needs some comments as well as updating from the older version of the lib)

4) General bug fishing and testing

---updates::

1) another speed upgrade for the 'prop(matrix, matrix)' function improved by ~30%

2) a full MPI controller class (instead of the C functions, we have an object now) with both BLOCKING and NON-BLOCKING commands

3) the 'biot' class has been parallelized for optimum performance

4) the 'solid' program (included in the examples) has been heavily updated and optimized for parallel running as well as a ton of other speed enhancements and minor bug fixes.

--version 0.8...

1) SPPPEEEEDDD -->

---Now allows the Fastest Fourier Transform in the West (FFTW)

<http://www.fftw.org>

library to be used as the main driver for Fourier Transforms of Vectors and Matrices...A very fast implementation for FFTs...

i highly recommend using this library...simply type

"--enable-fftw=<path to libs>" (like /usr/local/)

in the configure path, and everything will be properly set.

---the matrix Multiply code (for full matrices) has been
DRASTICALLY altered there is currently 2 setups possible...
as we all know, the machine specific BLAS as ALWAYS
much faster then anything a simple one man shows can create,
sooo i tries to create some 'okay' optimize code (meaning that
if you do not have a machine specific BLAS package, everything
still works, and with okay speed...ie.. it is still faster
then the basic netlib matrix multiply and many other packages,
but compared to what the vendors can give you ,
it will be ~2-3 times slower)...so if you want to use a vendor
BLAS package, you will need to 2 things

a) use the '--with-blas=<mylib>' option in the configure

b) Edit the top of 'container/matrix/matmatmul.h" to
reference the correct function within the lib

INTEL (LINUX)::***just so you know, the current set up in
'matmatmul.h' is for Intel's MKL library and ATLAS libs.
and will work on both CYGWIN and LINUX environments

NOTE-->I have found that the ATLAS tuned libraries
can be FASTER than the MKL libs on some Intel machines,
i invite you to try testing them versus each other.

Read the Windows.Readme for more info on how to configure
and use ATLAS for this library.

get ATLAS here

<http://math-atlas.sourceforge.net/>

<http://developer.intel.com/software/products/mkl>
specific for Pentiums (and it is quite speedy, in fact
i think it is a fast as can one can get) DO NOT use the
'lapack' lib or any Shared libs (i.e. no *.so) here,
use either the 'libmkl_p3.a' (for pentium 3's) or
'libmkl_def.a' for everything else

It requires the "pthread lib" on linux so do "--enable-pthread" in the configure line.....

you will need to set "OMP_NUM_THREADS" in your environment
if you have an SMP machine (careful, not setting this
can cause mass havoc if you also use MPI)

INTEL (CYGWIN/PC)::Read the "Windows.Readme" on how to make the
best BLAS lib and incorporate it into the library.
You will need 1) cygwin, 2) ATLAS...please read my
directions inside the windows readme file when
configuring ATLAS...

SGI:: go here for the lib optimized lib
<http://www.sgi.com/software/scsl.html>
(this has not yet been implemented in the library yet)

SUN:: go here for the lib optimized lib
http://docs.sun.com/htmlcoll/coll.118.3/iso-8859-1/PERFLIBUG/plug_bookTOC.html
(this has not yet been implemented in the library yet,
but SUN's library should come with the OS)

- 1a) Lots of other speed improvements have been made throughout the code making the Quantum Mechanical type simulation go ~5-10 times faster....(nice eh?)
 - 2) The rotation class has had 'Cartesian' or 'spherical' or 'both' types of rotations are not handled in an optimized way
 - 3) a bug fix for the second order Quadrupole interaction
- *****

--version 0.7...

- 1) The beginning of the parallel world...
some VERY BASIC MPI functions to allow one to compile code with or without the MPI library present...

-- In the works simple "get" and "put" methods for each container type (matrix, vectors, coords, etc) are being written...

-- NOTE:: this MPI implementation is not really designed for massively parallel systems (i.e. systems of 128 processors)
but more the 2-10 processors range where distributing loops is advantageous, the normal container operations (things like "vector + vector") will not be parallel unless you create a specific function to do so

(which should be simple once the main interface is complete)
look to the 'MPI' section of the documentation for more details and examples, and look to the 'INSTALL' section for the new configure options for MPI

- 2) a VNMR class called 'VNMRstream' that will read and write the binary formats (both 1D and 2D) from the VNMR (Varian) spectrometer system
 - 3) The general solid-state simulation application 'Solid' is now included as a glorified example in the 'examples' directory
 - 4) The very nice minimization package "MINUIT" from has now been included in the library using a heavily debugged and reworked 'C-MINUIT(<http://c-minuit.sourceforge.net/>)' to allow inclusion into the shared lib as well as C++ apps
- *****

--version 0.6

- 1) large shift in the way the blochparameters are treated
 - 2) able to have 3-vector offsets and off zaxis interactions
 - 3) general magnetic field calculator integrated with the offset characterization of a parameter space
 - 4) a "Parameters" class that make inputting of data config files very simple...
- *****

--version 0.5

- 1) Basic Quantum Mechanical backbone laid down
- 2) Spin and Spatial Tensors
- 3) Spin Systems
- 4) Optimized Solid state interaction classes
- 5) Power Average generator angle generator
- 6) Solid System (an extension to the Spin System)
- 7) Gamma-Compute algorithm for spinning FIDs
- 8) a simple one-D FID generator (spinning, or not)

--version 0.4

- 1) more documentation added....however, more classes and features added too..so it seems i am still in the same place as before...needing technical writer. any takers?
- 2) Stencils::vector stencils and there creation have been very easily implemented and finished...Stencils on Grid and associated data have been MOSTLY finished still lots of minor fixing and pieces to create, but the hard part is finished
- 3) spin tensors and spin systems (the standard Quantum Mechanical Types) have been added as auxiliary classes...(it beefs up the spin dynamics a bit)
- 4) Dipole-Dipole and Demagnitizing Field interactions have been added
- 5) Numerous fixes and improvements to main numerical engines...especially the BlochSolver...improved error checking and improved I/O control for multiple pulse sections/fid collection, etc
- 6) SHAPES have been mildly recast into expression template forms of '&&' and '||' (and/or) allowing precise control over multiple 'odd' shaped objects
(i.e. you can incorporate slice plane, cylinders and cubes in a simple 'plane && cylinder || cube' expression)
- 7) Edge detection created for shapes...to be used mostly for the Stencils (functional, but not refined) and Boundary conditions (main classes begun, not finished...need to create the Neumann BCs and a few other highly common BCs)..BC NOT integrated into anything yet...
- 8) more and more examples...from simple class tests to complex spin dynamical simulations...
- 9) HOLLY CHROME! it compiles on macs, linux, sgi's, and windows...this is too weird..lets hear it for gcc!

--version 0.2

- 1) started the large process of documentation
- 2) added 2 interactions to the "Interactions"
Bulk Susceptibility and Radiation Damping
- 3) allows for 'Dimensionless' Units for the magnetization
when solving the diff eqs (all spins Magnetization will Add to 1)

--version 0.1

- 1) basic data structures and algorithms lay out
Matrix, vectors, grids, coords, diffeq solvers, BlochParameters



The Distribution INSTALL helper file

```
#####
## 
## Bloch Lib Installation
##
## author Bo Blanton::magneto@dirac.cchem.berkeley.edu
## last update:: 8.20.02
#####
```

Platform Notes::

--It should compile on most any system with gcc installed on it
--to compile on windows machines you either need 'Cygwin'(gcc)
--I have only tried to compile this on Mac osX (and it works fine)
i do not know about os 9.1 or lower

Performance Notes::

--Compilation times for the fully optimized code can take a
VERY long time (~10-15 min for a single file)...Why? because of the
nature of the expression templates and the LARGE quantity of files
that require massive 'expressing' to function...
you can turn the optimizations off with the compilation commands

```
'{source-top}/blcc -g blaa.cc' --> for the installed library (done with 'make install')  
'{source-top}/blcc -n blaa.cc' --> for the local version (i.e. the place where it was untared)
```

which will greatly speed up compilation times
(and it quite helpful when debugging code...so you
do not have to sit and wait for years to debug things)

It is recommended that the final programs be fully optimized as you will see
~5-10 fold increase in speed using the commands

```
'{source-top}/blcc blaa.cc' --> for the installed library (down with 'make install')  
'{source-top}/blcc -nf blaa.cc' --> for the local version (i.e. the place where it was untared)
```

(for Codewarrior compilation there is an example
project in {source-top}/ide/blochlib_test.mcp...note
this is VERY out of date!!!! and does not work...)

**USEING THE LIBRARY (the 'blcc' shell script)

-- Because there can be oodles of different linked libraries
and include paths simply from the options you have chosen
from the library compilation, the compile command can be VERY long.
To get everything fully optimized and linked properly you'll need
to type in something like

```
c++ -O3 -fomit-frame-pointer -W -Wcast-qual -Wpointer-arith
```

```
-Wcast-align -pedantic -fstrict-aliasing -funroll-loops  
-finline-functions -ftemplate-depth=40 -mcpu=pentiumpro  
-I/home/magneto/code/blochlib-1.0/src/  
-I/home/magneto/include -I./  
-L/home/magneto/code/blochlib-1.0/src/.libs  
-L/home/magneto/lib <file>.cc -lbloch -fftw  
/var2/atlas/lib/libcblas.a /var2/atlas/lib/libatlas.a  
-lg2c -lm
```

(NASTY!!) SOOOO there is a little Shell script that does
'typing' for you...thus that compile command is reduced to
"blcc -nf <file>.cc"

*the Moral is to USE 'BLCC' !! otherwise your fingers will ache
*to see all the scripts options type
'blcc --help'

**To install...

- 1)go to {source-top} (something like "blochlib-0.8/")
- 2) run the config script with the an optimization options

--enable-profile
-->turns on profiling and debugging. disables the big optimization flags
--> this will disable the rest of the optimization options
"--with-gcc"
--> uses gcc as the compiler
"--with-gcc3"
--> uses gcc3 as the compiler
"--enable-debug"
--> turns on only debugging (snaps off the rest of the optimizations)
"--with-mpi=<mpi directory>"
--> turns on compilation of library using MPI
--> <mpi directory> will be something like
"/usr/local"
if the libs and includes are in /usr/local/include/
"--disable--minuit"
--> this will TURN OFF including MINUIT with the lib
(you only really need to do this if you do not have a fortran
compiler)
"--with-blas=<total path to library>"
-->Enables you to use your own platform specific BLAS library
which will be MUCH faster then the built in algos for this
lib (although the algorithms here have been optimized)
For now only the Matrix*Matrix algo is used
(I recommend going to ATLAS (<http://math-atlas.sourceforge.net/>)
and following my ATLAS directions in the "Windows.Readme" file)
--><total path to library> will be something like
"/usr/local/lib/libcblas.a"

--with-atlas=<atlas directory>
-->Enables you to use your own platform specific BLAS library
which will be MUCH faster then the built in algos for this
lib (although the algorithms here have been optimized)
For now only the Matrix*Matrix algo is used
(I recommend going to ATLAS (<http://math-atlas.sourceforge.net/>)
and following my ATLAS directions in the "Windows.Readme" file)
-->used to save typing from the '--with-blas' command
--><atlas directory> will be something like
"/usr/local/atlas/lib"
if the "libcblas.a" & "libatlas.a" are in that directory

--with-fftw=<total path to library>
-->Enables you to use the Fastest Fourier Transform in
the West (FFTW) libraries (<http://www.fftw.org>) to perform
FFTs on Vectors and Matrices

--enable-pthread
-->If any of the 'extra' libs you use require the pthread lib
this will turn it on (on LINUX with the INTEL, MKL specific BLAS
this is necessary)

--enable-exceptions
--> this turns on exception handling instead of the
standard 'exit' upon a critical error. This will make
the compile take longer to compile, AND the run time will suffer a
little bit, but for large applications, this may be necessary

example::

```
./configure --enable-profile  
./configure --with-gcc --enable-debug  
./configure --prefix=/usr/local/blochlib --with-mpi=/usr/local
```

**This is how I configure MY personal copy (this is **
**the best config possible...it uses the optimized ATLAS **
**libs, MPI, and FFTW for true computational zoom **

```
./configure --with-atlas=/home/magneto/atlas/lib --with-fftw=/home/magneto/fftw --with-  
mpi=/usr/local --prefix=/home/magneto
```

- 3) 'make'
- 4) 'make install'

```
*****  
* Unless you have not already done so, you should first *  
* make the Profile library, the Debug library, and *  
* LASTLY make the fully optimized library. So here *  
* would be the correct ordering for the correct building *  
* *  
* ****To use MPI**** *  
* --with-mpi=<path to libs and includes> *  
* *
```

```

* ****To use Exceptions***** *
* --enable-exceptions *
*
* ****To DISABLE MINUIT (for those folks with no fortran) *
* --disable-minuit *
*
* ****To Use your optimized/platform specific BLAS Library *
* --with-blas=<total path to library> *
* --enable-pthread (necessary on LINUX with INTELs MKL) *
* --with-atlas=<path to ATLAS libs> *
*
* ****To Use the FFTW Fast Fourier Transform Libs *
* --with-fftw=<total path to library> *
*
* ./configure --enable-profile --prefix=<your install dir> *
* make *
* make install *
*
* ./configure --enable-debug --prefix=<your install dir> *
* make clean *
* make *
* make install *
*
* ./configure --prefix=<your install dir> *
* make clean *
* make *
* make install *
*
* This will ensure you have all the tools you need to write *
* and TEST your programs for speed and give you the best *
* speed when you have finished your profiling or debugging *
*
* There is a script file called 'blcc' that takes care of *
* linking the correct library and including the correct *
* include paths...(i.e. it uses the profile lib, the debug *
* lib, or the optimized lib) *
* this file can be found in this directory..... *
*****

```

The lib is stored in the "src/.libs/" directory
the master header file "src/blochlib.h" if
you did not run 'make install'

--right now the build process is quite basic...

--the generated library "libbloch.a" should be linked like so

g++ -I{path to 'blochlib.h'} -O3 {other optimization flags} <Your file.cc> -L{path to libbloch.a} -lbloch

--To compile the examples in the "examples/" directory do this

--NOTE: you MUST include the flag "-ftemplate-depth-40" or the compilation will barf

```
g++ -O3 -finline-functions -funroll-loops -ftemplate-depth-40 -I./src/ <example.cc> -L{source-top}/src/.lib/ -lbloch -o <prog name>
```

or use the inhouse script

```
cd {source-top}/examples/  
{source-top}/blcc -nf <example.cc>
```

OR if you have installed the library with 'make install'

```
{source-top}/blcc <example.cc>
```

Getting BlochLib To work On Windows

**** Getting BlochLib To work On Windows
**** Getting OPTIMIZED BLAS routines on Windows
**** (also getting BlochLib to work as well)

author: Bo Blanton
last update: 06.20.02
email: bo@theaddedones.com

*This document will hopefully show you how to create a highly optimized set of BLAS libraies for the CYGWIN and MINGW environments.

-----Note for Visual C++ -----

!!THIS DISTRIBUTION IS NOT SET UP TO COMPILE ON ViSuAl Cpp!!

(it should, i just do not have it to try it)

If you are using Visual C++ then INTEL has already created a very fast set of BLAS routines optimized for each Processor type.
Go here to get the DLL and static libs..

<http://developer.intel.com/software/products/mkl/index.htm>
the linking directions are in that distribution

-----CYGWIN & MINGW -----

*CYGWIN and MINGW are unix emulation layers over Windows that allow folks to write code that will copmile on Linux, Unix, and Windows with minimal effort.

*CYGWIN & MINGW--> CYGWIN provides a POSIX emulation layer over windows so one can use all the good UNIX command line tools and scripting capabilities (like bash, perl, etc)...it provide the GCC compiler suite as well. However, all programs and/or libaraies are dependant on the Cygwin Run time DLL...meaning 'double cliking' on a program will typically result in an error becuase it cannot find the cygwin dll thus to run the program you must have cygwin installed...which is not bad so long as you are not going to distribute the programs. MingW solves this problem, by linking 'reverting' the basic cygwin dependancies on to the Windows Sub System (the OS). MINGW is simply a new set of libraries and includes to replace the Cygwin ones...cygwin, however, allows the usual make and bulid process to happen flawlessly...so you need both...

*Getting CYGWIN::

- 1) go here: <http://cygwin.com>
- 2) run the 'setup.exe' program and read the directions to install (i think that is easy enough)
- 3) the installation will typically be in "C:cygwin" but you

can change it...

*Getting MINGW runtime libs

- 1) go here: <https://sourceforge.net/projects/mingw/>
 - 2) get the newest release (as of this writing it was "mingw-1.1")
 - 3) open the Cygwin Bash Shell....
 - 3) create a sub directory "C:cygwinmingw"
 - 4) copy "MinGW-1.1.tar.gz" to "C:cygwinmingw"
 - 5) untar the dist. "tar -zxvf MinGW-1.1.tar.gz"
- below the copyable commands

```
mkdir /mingw
cp MinGW-1.1.tar.gz /mingw
tar -zxvf MinGW-1.1.tar.gz
```

*You should now have everything you need to start compiling things in any way, shape, or form you desire.

now Read the "README.BLAS" for getting really fast BLAS routines on your particular system....

Getting OPTIMIZED BLAS routines on your system

**** Getting OPTIMIZED BLAS routines on your system
**** (also getting BlochLib to work fast as well)

author: Bo Blanton
last update: 04.23.02
email: bo@theaddedones.com

-----ATLAS - Fast AutoMatic BLAS-----

*BLAS (Basic Linear Algebra) tend to be the backbone of most numerical computations, and tend to be where most of the CPU time is spent. Each Processor type has its own set of optimizations (things like L1 cach sizes, alignments, L2 size, etc) Because of each processor's different optimizations, creating highly optimized numerical routines is a difficult task...for instance unoptimized (yet simply coded) matrix multiplications can take up to 10-20 longer then the optimized versions...the optimized versions, however, are very hard to write...

*ATLAS is a project designed to create FAST BLAS on most CPU architectures. Using several timers and outcode generation it should find you some of the best routines around.

*NOTE::If the vendor has created its own set of BLAS libs, then they CAN be faster then ATLAS's, but i have found for INTEL systems ATLAS is better.

*Getting ATLAS:

- 1) go here: <http://math-atlas.sourceforge.net/>
- 2) download the latest version (i used 3.3.14)

*The Set UP::

--This assumes that the Cygwin is installed on Windows or that you are using gcc of some kind
-- if you are using this on windows

- 1) Open up a shell
- 2) down load atlas
- 3) untar the file "tar -zvxf atlas.3.3.14.tar.gz"
- 4) cd into "ATALAS"
- 5) type 'make'

tar -zvxf atlas.3.3.14.tar.gz
cd ATLAS
make

- * This will then take you to the interactive ATLAS questions
- * Below you will find the 'good' answers' to the
- * ATLAS set up, everything that i do not tell you an answer here,
- * simply choose the default

- choose your OS to be what ever it is
- and your appropriate processor
- to "enable Posix threads support?" say NO
- choose *NOT* to use "express Set up"
- at the "Enter Maximum cache size(KB):" question
enter 2 times the 'default' amount!!
- For the Fortran compiler enter "g77"
do NOT use the default "/usr/bin/g77.exe"
- For the C compiler enter "gcc"
do NOT use the default "/usr/bin/gcc.exe"
- For the C compiler for generated code enter "gcc"
do NOT use the default "/usr/bin/gcc.exe"
- For "Enter Ranlib" : enter in "ranlib"

****REALLY IMPORTANT (see will be compromised)***

- for the question "Use supplied default values for install" say *NO*

- After the configuration has been done type

make install arch=<your arch here>

where <your arch here> for a Pentium III on windows would be something like "WinNT_PIII"

and wait a long while (~hour or more even 7 for AMD chips)
as it goes through the testing and creation

- when it is finished go to the 'ATLAS/lib/' directory
- find the directory with your <arch> name
something like "ATLAS/lib/WinNT_PIII/"

**You need to use 2 libs to get the linking to function properly
**Within 'blochlib' and associated compilations IN THIS ORDER!!

to use in BlochLib you need to use the quotes

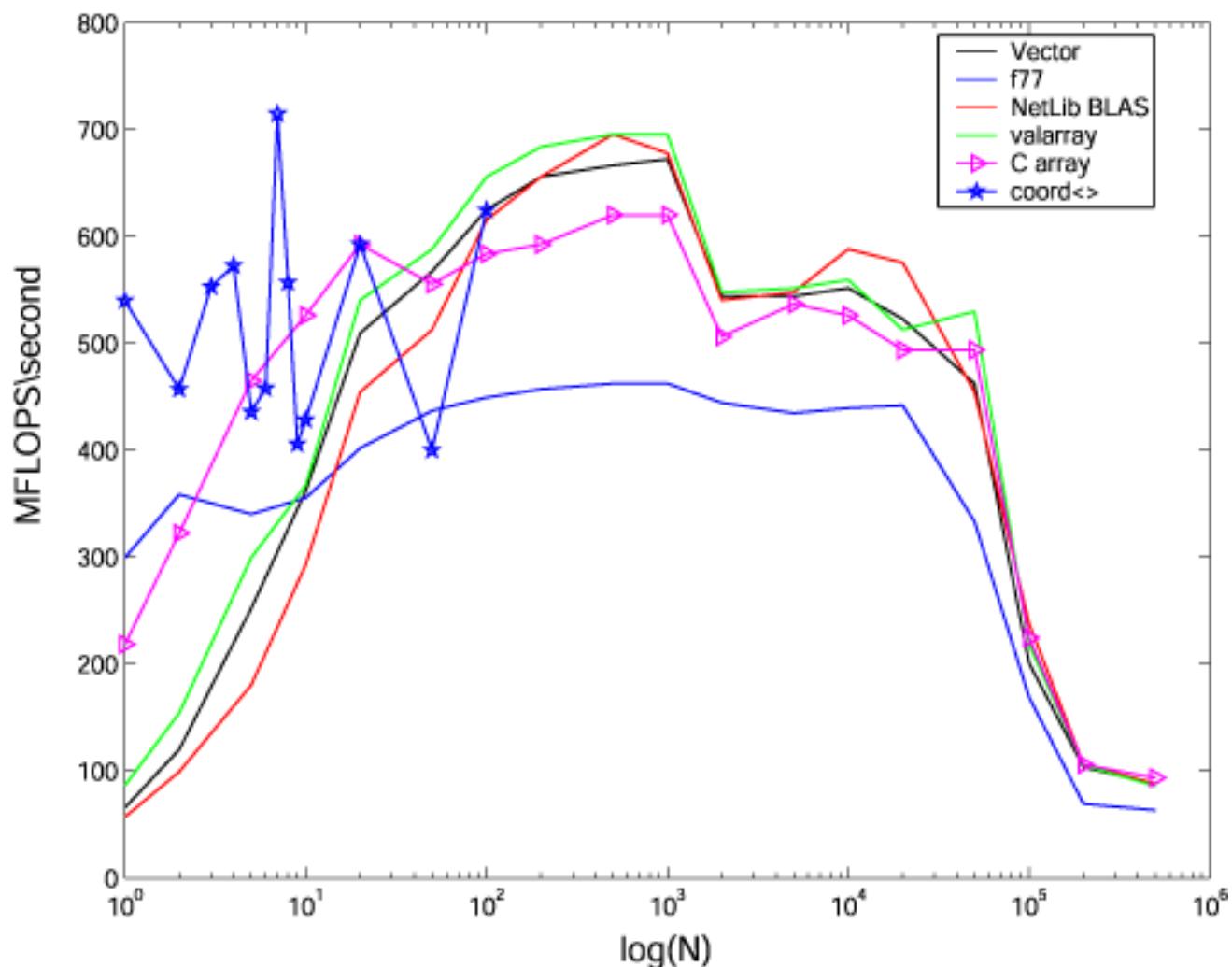
--with-blas="<prepath>/ATLAS/lib/<arch>/libcblas.a <prepath>/ATLAS/lib/<arch>/libatlas.a"

OR the simpler one
--with-atlas="<prepath>/ATLAS/lib"

Speed Tests

- Below are simple speed comparison tests of representative math operations for the this library and other library packages, as well the 'quick-and-dirty' simple code most people would program.
- These tests were all performed using gcc 2.95.3 as the compiler with the optimization options given in the "blcc" compilation shell script (-O3 -funroll-loops -finline-functions -mpentiumpro, etc).
- All tests were run on a 700 MHz Pentium III Xeon (L2=2Mb, L1=16Kb) processor.
- ATLAS was configured as given in the README.BLAS file
- another common NMR library '[Gamma](#)' was compiled used as is.
- Vectors...

DAXPY($a+=\text{const} \cdot b$) 700 MHz Pentium III Xeon

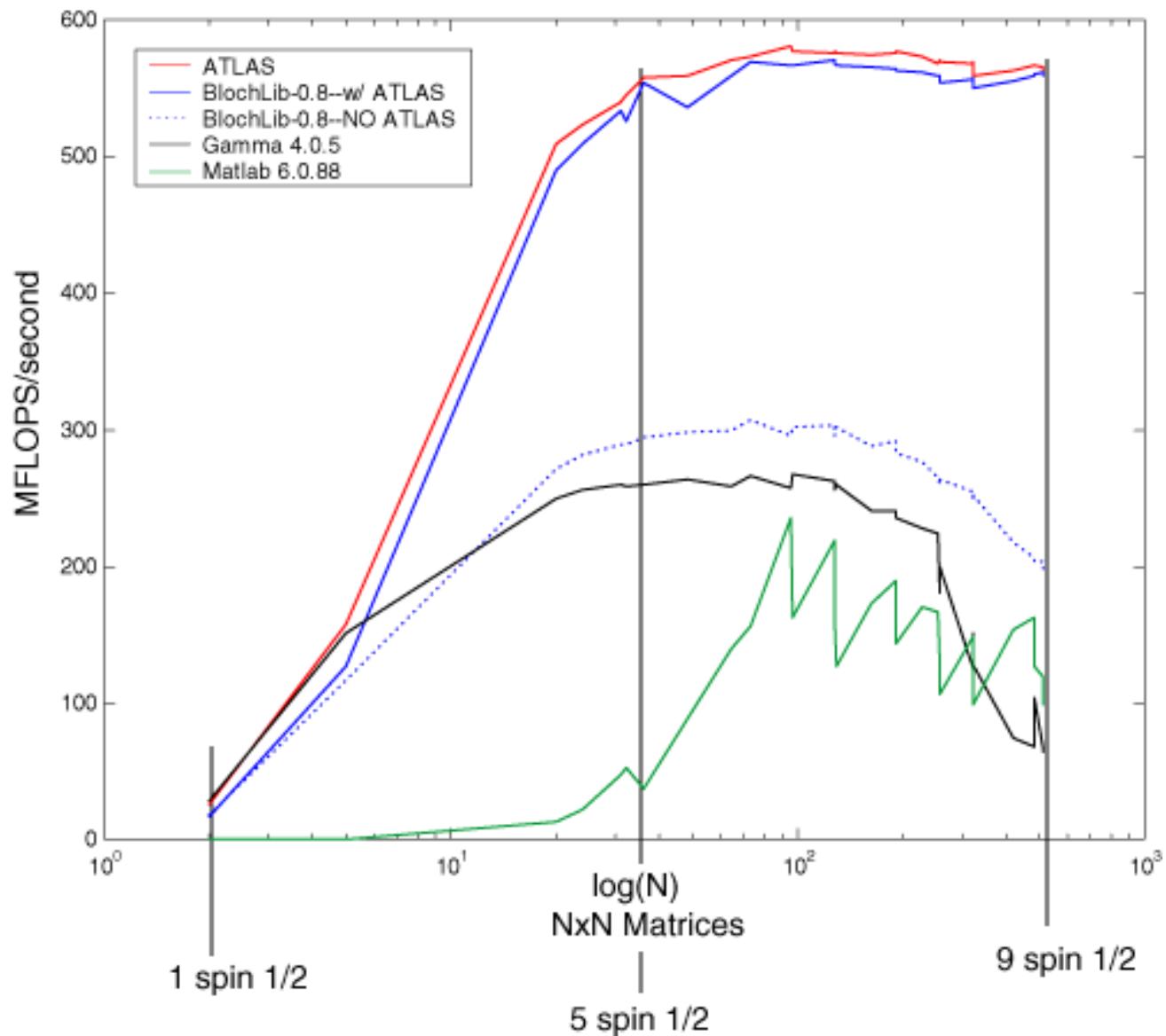


- Propagators Matrix Multiply

Full Complex Matrix Speed Tests

c=a*b*adjoint(a)

700 MHz Pentium III Xeon/Linux



- This script makes compilation of your programs using the lib quite simple...it supports a number of options for different modes and configurations...it is basically here to save you (and me) from typeing the "c++ -O3 -f...very long a nasty compile string" into the console everytime you want to compile something.
- The script is a standard Shell script and will be installed into "{your install directory}/bin/"

- **The 'blcc' Options**

```
BlochLib Version 1.0 Compile Script Usage
standard C++ optimizations and other parameters are allowed
theses options below make easy flags to compile using the most
common methods.

NOTE:: using the script with no options will use the maximal
optimizations and the INSTALLED version.

NOTE:: the options begining with '-n' refer to the LOCAL/untared
directory (typically the root of the compilation set)

{option} ..... {meaning}
?, -h, --help ..... this message
-v, --version ..... version info
-g ..... compile using INSTALLED debug version
(libblock_g)
-p, -pg ..... compile using profiled INSTALLED version
(libblock_pg)
-m ..... compile using MINIMAL optimizations for
INSTALLED version
..... (fast compile for syntax debugging)
-mpi ..... compile using FULL optimizations for
..... INSTALLED version that has been compiled with
MPI
-n ..... compile using MINIMAL optimizations for LOCAL
version
..... (fast compile for syntax debugging)
-nf ..... compile using FULL optimizations for LOCAL
version
-ng ..... compile using LOCAL debug lib version
-np ..... compile using LOCAL profiled lib version
-nmpi ..... compile using LOCAL lib version with MINIMAL
optimizations
..... and that was compiled with MPI
..... (fast compile for syntax debugging)
-nfmpi ..... compile using LOCAL lib version with FULL
optimizations
..... and that was compiled with MPI
-ngmpi ..... compile using LOCAL lib version with DEBUGGING
optimizations
..... and that was compiled with MPI
```

- Some Examples::

- `blcc {myfile}.cc -o {outname}`

---compiles 'myfile.cc' into an executable 'outname'
 ---this is simplest command to use BUT IT REQUIRES that you have performed a 'make install'
 --- This also runs the compilation with the MAXIMAL OPTIMIZATIONS (this determined based on your machine...)...these optimizations can make your program run ~10 times faster than the non optimized counter part BUT it also requires ~10 times long to compile...this library being mostly header files requires abnormally long compilation times...it is a price to pay for speed...
- `blcc -m {myfile}.cc -o outname`

---compiles 'myfile.cc' into an executable 'outname'
 --- IT REQUIRES that you have performed a 'make install' (looks for library files in the installed dir)
 ---This runs the compilation with the MINIMAL OPTIMIZATION...results in fast compile but slower code...Useful for initial debugging of code where speed is not the major concern (getting things to work is...)
- `blcc -g {myfile}.cc -o {outname}`

---compiles 'myfile.cc' into an executable 'outname'
 --- IT REQUIRES that you have performed a "./configure --enable-debug" AND 'make install' (looks for library files in the installed dir)
 ---This runs the compilation with the DEBUGGING INFO...it sets all vector and matrix bounds checking ON and compiles code so it can be used in typicall debuggers....
- `blcc -pg {myfile}.cc -o {outname}`

---compiles 'myfile.cc' into an executable 'outname'
 --- IT REQUIRES that you have performed a "./configure --enable-profile" AND 'make install' (looks for library files in the installed dir)
 ---This runs the compilation with the DEBUGGING INFO AND PROFILING INFO...it sets all vector and matrix bounds checking ON and compiles code so it can be used in typicall debuggers....and with 'gprof'
- `blcc -mpi {myfile}.cc -o {outname}`

---compiles 'myfile.cc' into an executable 'outname' coupling the libraries of MPI (the Message Passing Interface...make thing parellel) ...
 ---NOTE:: none of the code in the lib is explicitly parellel yet...however there are certain codes you could write easily to use MPI alibility..this is also here becuase soon there will be mpi built into some of the codes...
 --- IT REQUIRES that you have performed a 'make install' (looks for library files in the installed dir)
 --- IT REQUIRES MPI
 ---This runs the compilation with MPI libs linked into the file...compiles at FULL

optimizations

- - blcc -n {myfile}.cc -o {outname}
 - blcc -nf {myfile}.cc -o {outname}
 - blcc -nmpi {myfile}.cc -o {outname}
 - blcc -nfmpi {myfile}.cc -o {outname}
 - blcc -ng {myfile}.cc -o {outname}

---compiles 'myfile.cc' into an executable 'outname' BUT uses the initial UNTARED directory as the source for the libraries...

"-n" --- MINIMAL OPTIMIZATIONS

"-nf" --- FULL OPTS

"-nmpi" --- mpi with minimal opts

"-nfmpi" --- mpi with full opts

"-ng" --- debugging compilation

---this is most usefull mostly for me as to do a make install everytime just get tedious when writting new things...

--Constructors class Matlab5{....
--- matstream or
--- matstream class matstream{...

--Element Extraction

--- get

--IO

--- close

--- is_open

--- open

--- put

--Other Functions

--- whos

Examples

--- read/write

This is a general Matlab Version 5 (or greater) Reader and writer....It writes matlab binary files that can be easily read into matlab for further processing...it can also read structures from the matlab v5 or greater files.

I'd like to thank the mathworks (makers of matlab) folks for there nice description of the "MAT-file" and "Gamma" for filling in some tricky 'compression' items...

Matlab::constructor

Function: **matstream()**

Input: none

Description: empty constructor--> Sets up the default matlab stream
iomode=ios::out | ios::binary

Example Usage: matstream moo;
moo.open("infile"); //opens a file after declaration

Matlab::constructor

Function: `matstream(std::string infile, int iomode=ios::binary | ios::out)`

Input:

- infile-->the file you wish to begin extracting all the matlab
- iomode--> set the input/out mode

Description:

- will either open a file for writing or reading....

Example Usage: `matstream moo("myfile.txt" , ios::binary | ios::out);`

Matlab::element extraction

Function:

- `void get(std::string varname, int &var)`
- `void get(std::string varname, double &var)`
- `void get(std::string varname, complex &var)`
- `void get(std::string varname, Vector<int> &var)`
- `void get(std::string varname, Vector<double> &var)`
- `void get(std::string varname, Vector<complex> &var)`
- `void get(std::string varname, rmatrix &var)`
- `void get(std::string varname, matrix &var)`

Input:

- `varname`--> the name of the variable within a matlab file...
- `var`--> the variable you wish to fill from the matlab file...the overloads you see above are the types allowed

Description:

- fills the "var" from the data in the matlab file...assuming that "varname" exists inside the file..

Example

- `matstream inmat("data.mat", ios::binary | ios::in);`
- `Vector<complex> vdat;`
- `inmat.get("vectord", vdat);`

Usage:

Matlab::IO

Function: **void close()**

Input: none

Description: closes the file...releases the file handel so you can manipulate it elsewhere if you deisre

Example Usage: matstream myout;
myout.open("input.mat");
complex num(4,5);
myout.put("cpnum" , num);
myout.close();

Matlab::IO

Function:

□ **bool is_open()**

Input:

□ none

Description:

□ returns true if the file is open...false if not.

Example Usage:

```
□ matstream myout;  
myout.open("input.mat");  
complex num(4,5);  
myout.put("cpnum", num);  
if(myout.is_open()) myout.close();
```

Matlab::IO

Function: `void open(const std::string fname, int iomode=ios::binary | ios::out)`

Input: `fname`--> file name to open...
`iomode` --> the mode for the opening...

Description: reads the file into the object

Example `matstream myout;`

Usage: `myout.open("input.mat");`
`complex num(4, 5);`
`myout.put("cpnum" , num);`

Matlab::IO

- Function:**
- `void put(std::string varname, int &var)`
 - `void put(std::string varname, double &var)`
 - `void put(std::string varname, complex &var)`
 - `void put(std::string varname, Vector<int> &var)`
 - `void put(std::string varname, Vector<double> &var)`
 - `void put(std::string varname, Vector<complex> &var)`
 - `void put(std::string varname, rmatrix &var)`
 - `void put(std::string varname, matrix &var)`
 - `void put(std::string varname, Vector<coord<NumT, N>> &var)`
 - `void put(std::string varname, _matrix<coord<numt, N>, Strucutre>&var)`
- Input:**
- varname--> the variable name you wish to call the out data in MATLAB
 - var--> the variable you wish to write to the file...
- Description:**
- puts the data into the file...
- The "Vector<coord<NumT, N>>" and "_matrix<coord<NumT, N>, Structure>" get written specially.
for each direction (from 1...N) a sepearate Array is written so
for a Vector<coord<double, 3>>
there will be three separate vectors in matlab called "varname"+0, "varname"+1, "varname"+2
the same thing will happen for the matrix brands...
- Example**
- `matstream myout;`
- Usage:**
- `myout.open("input.mat");`
 - `complex num(4,5);`
 - `myout.put("cpnum" , num);`
 - `Vector<complex> outvec(45, num);`
 - `myout.put("veccpnum" , outvec);`
 - `_matrix<coord<>, FullMatrix> matout(5,5,coord<>(3,2,1));`
 - `myout.put("coordmat" , matout);`
 - `myout.close();`

Matlab::other

Function: `void whos(ostream &out=std::cout)`

Input: `out-->` another output stream to dump the "whos" data to (default=`cout`)

Description: `prints out the information of the items inside a Matlab file exactly like the "whos" command does in matlab itself`

Example `matstream myout;`

```
myout.open( "input.mat" );
complex num(4,5);
myout.put( "cpnum" , num );
Vector<complex> outvec(45, num);
myout.put( "veccpnum" , outvec );
_matrix<coord<>, FullMatrix> matout(5,5,coord<>(3,2,1));
myout.put( "coordmat" , matout );
myout.close();

myout.open( "input.mat" , ios::binary | ios::in );
myout.whos();

//this will print

//Name Size Bytes Class
//cpnum 1x1 16 double precision array (complex)
//veccpnum 45x1 720 double precision array (complex)
//coordmat0 5x5 200 double precision array
//coordmat1 5x5 200 double precision array
//coordmat2 5x5 200 double precision array
```

matlab::reading and writing

There are essentially 2 functions you need to know... get (to read) and put (to write)

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc,char* argv[]){
//get the file name of the matlab file
    int q=1;
    std::string fname;
    query_parameter(argc,argv,q++, "Enter File Name to read: ", fname);

//open a matlab file for reading
    matstream mat(fname, ios::in | ios::binary);
//print to the console what is in the file...
//(this prints things exactly like 'whos' in matlab itself)
    mat.whos(cout);

//get a complex vector from the inputfile named 'vdat'
    Vector<complex> loo;
    mat.get("vdat", loo);

//get the simple variable 'zfill' from the file
    double zf;
    mat.get("zfill", zf);

//close the input file...
    mat.close();

//open a file for output...
    matstream outt("out.mat", ios::out | ios::binary);

//define several vars to write...
    complex poo(7,8), hhh(3,4);
    Vector<complex> outmat(loo.size(), complex(6,6));

//put these containers giving them matlab names...
    outt.put("vdat", outmat);
    outt.put("poo",poo);
    outt.put("hhh",hhh);

    Vector<complex> hj(7, complex(3,2));
    Vector<int> kj(5,4);

//puts matrices, vectors, numbers, etc..
    outt.put("loo", loo);
```

```
    outt.put("hj",hj);
    outt.put("kj",kj);
    outt.put("jjj",kj);
    matrix kji(5,5,3);
    outt.put("kji", kji);

    //put a more complex data strcuture into the file...
    //the output in the matlab file will be 'VecC0', 'VecC1', VecC2'
    //corresponding to each direction in the coord (x,y,z)
    Vector<coord>> jk(35, 21);
    outt.put("VecC", jk);

    //close the output file
    outt.close();

    //open the ouput file and check output its contents to the console
    matstream mat2("out.mat", ios::in | ios::binary);
    mat2.whos(cout);
    //the whos command should print out...

    //Name Size Bytes Class
    //vdat 10x1 160 Double Precision Array (complex)
    //poo 1x1 16 Double Precision Array (complex)
    //hhh 1x1 16 Double Precision Array (complex)
    //loo 10x1 160 Double Precision Array (complex)
    //hj 7x1 112 Double Precision Array (complex)
    //kj 5x1 40 Double Precision Array
    //jjj 5x1 40 Double Precision Array
    //kji 5x5 400 Double Precision Array (complex)
    //VecC0 35x1 280 Double Precision Array
    //VecC1 35x1 280 Double Precision Array
    //VecC2 35x1 280 Double Precision Array

}


```

--Global Constants Global Utility Functions and Constants

[--- DEG2RAD](#)[--- GAMMA1H](#)[--- hbar](#)[--- HZ2GAUSS](#)[--- kb](#)[--- No](#)[--- permVac](#)[--- pi](#)[--- PI2](#)[--- RAD2DEG](#)**--Global Functions**[--- BinaryReadString](#)[--- BinaryWriteString](#)[--- BLEXCEPTION](#)[--- collapsVS](#)[--- dbtost](#)[--- itost](#)[--- max](#)[--- min](#)[--- parse_param](#)[--- parse_param](#)[--- query_parameter](#)[--- sign](#)

A seemingly random set of functions that are used for those random bits and pieces needed throughout the lib...

Utils/Constants::global constant

Function:

DEG2RAD

Input:

none

Description:

Degrees to Radians conversion (Pi/180)

Example Usage:

double moo=DEG2RAD;

Utils/Constants::global constant

Function: **GAMMA1H**

Input: none

Description: the gamma factor (gyromagnetic ratio in RADIAL frequency units) for a Proton....26.7519E+7....

Example double moo=GAMMA1H;
Usage:

Utils/Constants::global constant

Function: **hbar**

Input: none

Description: planks constant/2*Pi ...1.05E-34...in units of Joule*seconds

Example Usage: double moo=hbar;

Utils/Constants::global constant

Function:

HZ2GAUSS

Input:

none

Description:

conversion from Hz to Gauss...0.714567e-6...

Example Usage:

double moo=HZ2GAUSS*300.0;

Utils/Constants::global constant

Function:

kb

Input:

none

Description:

Boltzmann's constant ...1.3E-23...in units Joule/Kelvin

Example Usage:

double moo=No;

Utils/Constants::global constant

Function:

Input:

Description:

Example Usage:

- No**
- none
- Avagadros number...6.02E23
- double moo=No;

Utils/Constants::global constant

Function:

permVac

Input:

none

Description:

permefitvity of vaccume... $\text{Pi} \times 4.0 \times 10^{-7}$ in units of ($\text{T}^2 \text{ m}^3/\text{J}$)

Example Usage:

double moo=permVac;

Utils/Constants::global constant

Function: **Pi, pi, PI**

Input: None

Description: the number Pi...3.1415926535897932384626433832795

Example Usage: double moo=PI;

Utils/Constants::global constant

Function: **double PI2**

Input:

Description: $2\pi \dots 6.283185307179586476925286766559$

Example Usage:

Utils/Constants::global constant

Function:

RAD2DEG

Input:

none

Description:

Radians to Degrees conversion (180/Pi)

Example Usage:

double moo=RAD2DEG;

Utils/Constants::global function

Function: **std::string BinaryReadString(fstream &in, int len)**

Input: out--> a BINARY file to read the from

len--> the number of characters to read from the files

Description: reads the string in binary format from the fstream "in"...if it fails it returns "" (NULL string)

Example Usage: std::string moo;
fstream in("myfile", ios::in | ios::binary);
moo = BinaryReadString(in, 10);

Utils/Constants::global function

Function: **bool BinaryWriteString(std::string st, fstream &out)**

Input: out--> a BINARY file to read the from

 len--> the number of characters to read from the files

Description: reads the string in binary format from the fstream "in"...if it fails it returns "" (NULL string)

Example Usage: std::string moo;
 fstream in("myfile", ios::in | ios::binary);
 moo = BinaryReadString(in, 10);

Utils/Constants::global function

Function: □ **BLEXCEPTION(const char * filename, int line)**

Input: □ filename--> the file name. You should use __FILE__ (the C macro that defines the current file).
line--> the line number in the file. You should use __LINE__ (the macro that defines the current line).

Description: □ This is a global Macro. If you choose to compile with exceptions (i.e. the --enable-exceptions in the configure script) this macro will THROW a BL_exception() class object (the blochlib exception class). If you did not compile with exceptions, it will simply raise a SIGINT signal. This will either exit the program, or, if you have redefined the SIGINT handle function, will call that function (see "signal.h" man files in the standard C world of things).

Example □ `//a simple 'leave' statement`

Usage: `if(3!=4){
 BLEXCEPTION(__FILE__, __LINE__)
}`

Utils/Constants::global function

Function: `std::string collapsVS(Vector<std::string> &in);`
`std::string collapsVS(Vector<std::string> &in, int lim2);`
`std::string collapsVS(Vector<std::string> &in, int lim1, int lim2);`

Input: in-->A Vector of strings that you wish to collapse down to a single string
lim2--> the ENDing index to collapse
lim1--> the STARTing index

Description: -Collapses a vector of strings into one long string...
--the first function defaults as lim1=0, lim2=END;
--the second function defaults as lim1=0
--the third function has no defaults

Example `char *moo="par b, par looo, for r";`

Usage: `Vector<std::string> pars=parse_param(moo, ',');`
`cout<<pars[0]; //will print "par b"`
`cout<<pars[1]; //will print "par looo"`
`cout<<pars[2]; //will print "for r"`
`std::string colls=collapsVS(pars); //returns "par b, par looo, for r"`
`colls=collapsVS(pars, 1); //returns "par b, par looo";`
`colls=collapsVS(pars, 1, 2); //returns "par looo, for r";`

Utils/Constants::global function

Function: `std::string dbtost(double d, const std::string& fmt)`

Input:

- `fmt`--> the standard 'sprintf, printf' type formatting (things like %s, %d, etc)
- `d`--> the double to convert to a formatted string

Description:

- Returns a string from the double with the format given by `fmt`

Example Usage:

- `double g=956.951321516780;`
- `std::string oo=dbtost(g, "%6.2f"); //oo is "956.90";`

Utils/Constants::global function

- Function:** `std::string itost(int i,const std::string &fmt="%d")`
- Input:** `i--> an integer`
`fmt--> the normal 'printf' type C format style (like "%d")`
- Description:** `Returns a string from an integer with the format given by fmt.`
- Example Usage:** `int i=990;`
`std::string oo=itost(i); //oo is "990";`
`std::string oo2=itost(i, "%d"); //same as above`

Utils/Constants::global function

Function:

- `template<class T>`
`T max(T in, T in2)`

Input:

- `in`--> a numerical class
`in2`--> another number (or item with comparable pieces)

Description:

- returns the max of the two numbers (`in` or `in2`)

Example Usage:

- `double moo=-34, moo2=78;`
`double ss=max(moo, moo2); //ss=78`

Utils/Constants::global function

Function:

- `template<class T>`
`T min(T in, T in2)`

Input:

- `in`--> a numerical class
`in2`--> another number (or item with comparable pieces)

Description:

- returns the min of the two numbers (`in` or `in2`)

Example Usage:

- `double moo=-34, moo2=78;`
`double ss=min(moo, moo2); //ss=-34`

Utils/Constants::global function

Function: □ `Vector<std::string> parse_param(char * in)`
□ `Vector<std::string> parse_param(std::string in)`

Input: □ `in`--> A longish string of text (typically inputed from a text file)

Description: □ Creates and returns a vector of strings where each entry in the vector is obtained by splitting up the "char *" (or a string) by WHITE SPACE separations

Example □ `char *moo="par b";`

Usage: □ `Vector<std::string> pars=parse_param(moo);`
`cout<<pars[0]; //will print "par";`
`cout<<pars[1]; //will print "b";`

Utils/Constants::global function

Function: □ `Vector<std::string> parse_param(char * in, char which)`
 `Vector<std::string> parse_param(std::string in, char which)`

Input: □ `in`--> A longish string of text (typically inputed from a text file)
`which`-->the char you wish to chop up the input string

Description: □ Creates and returns a vector of strings where each entry in the vector is obtained by splitting up the "char *" (or string) by WHICH separations

Example □ `char *moo="par b, par looo, for r";`

Usage: `Vector<std::string> pars=parse_param(moo, ' ',');`
 `cout<<pars[0]; //will print "par b"`
 `cout<<pars[1]; //will print "par looo"`
 `cout<<pars[2]; //will print "for r"`

Utils/Constants::global function

Function: `void query_parameter(int argc,char* argv[],int par,const std::string& Q, T &V)`

Input: argc-->the count of the console input to the program

argv-->the console input to the program

par--> which parameter you wish to obtain (STARTS AT 1)

Q--> The output message if the argc<par

V--> THe variable you wish to assign the input value (can be.. char *, int, double, string)

Description: sets and obtains the parameter from the console when the program is run

Example `int main(int argc,char* argv[]){`

Usage: `int q=1;`

`double mypar=0;`

`query_parameter(argc, argv, q++, "Enter in the Desired value for mypar", mypar);`

`cout<<mypar<<endl;`

`return 1;`

`}`

Utils/Constants::global function

Function:

□ **template<class T>**
 T sign(T in)

Input:

□ in--> a numerical class

Description:

□ returns the Sign of the number in...
 if in ≥ 0 returns 1
 else in < 0 returns -1;

Example Usage:

□ `double moo=-34;`
 `double ss=sign(moo); //ss=-1`

--Constants

--- resolution

--Constructors

--- timer()

--- timer()

--Element Extraction

--- operator()

--Other Functions

--- reset

Examples

--- using timer

```
class Timer { ...
```

This small class acts as a small simple timer for programs...

it is not terribly precise (i.e. the smallest resolvable division is ~10e-6 seconds)...

it has 2 modes a 'HighRes' timer which has a resolution of ~1 micro second, but this timer can 'overflow' (meaning it cannot keep track for too long) after about an hour (depending on the MHz of your machine)

the second mode 'LowRes' has a resolution of only 1 second, but can be track of time for many days/months/years without overflow.

its usage is quite simple...

Timer::constant

- Function:** **timer_type resolution**
- Input:**
- Description:** The timer's resolution can be either
timer::HighRes (for ~1e-6 sec res, but only for ~1 hour)
timer::LowRes (for 1 sec res, but no overflow)
- Example Usage:** `timer myT;
myT.resolution=timer::LowRes;
myT.reset();`

Timer::constructor

Function: **timer()**

Input: none

Description: constructor...

sets the default Resolution to 'HighRes'

Example timer stopwatch; //make the timer global...as soon as you declare
Usage: it...it starts recording the time..

```
int main(int argc,char* argv[]){
    initialization stuff....  

    ....  

    stopwatch.reset(); //resets the timer to 0 so we can time the hard  

    calculation rather then the intialization  

    ...begin Hard calcualtion...  

    ...end Hard caclualtion...  

    cout<<"Time Taken: "<<stopwatch()*1e3<<" ms "<<endl; //print out  

    the time...
}
```

Timer::constructor

Function: □ `timer(timer::timer_type myRes)`

Input: □ `myRes-->` the timer resolution (`timer::HigRes`, `timer::LowRes`)

Description: □ A constructor that give you the ability to set the timer resolution at its creation

myRes can be

`timer::HighRes`, or `timer::LowRes`

Example □ `timer stopwatch(timer::LowRes); //make the timer global...as soon as you declare it...it starts recording the time..`

```
int main(int argc,char* argv[]){
initialization stuff.....
....
stopwatch.reset(); //resets the timer to 0 so we can time the hard
calculation rather then the intialization
...begin Hard calcualtion...
...end Hard caclualtion...

//print out the time...
cout<<"Time Taken: "<<stopwatch()<<" s"<<endl;
}
```

Timer::element extraction

Function: `double operator()() const`

Input: `none`

Description: `returns the "current" time`

Example `timer stopwatch; //make the timer global...as soon as you declare it...it starts recording the time..`

```
int main(int argc,char* argv[]){
// initialization stuff....
// ....
stopwatch.reset(); //resets the timer to 0 so we can time the hard
calculation rather then the intialization
// ...begin Hard calcualtion...
// ...end Hard caclualtion...

cout<<"Time Taken: "<<stopwatch()*1e3<<" ms "<<endl; //print out
the time...
}
```

Timer::other

Function:

□ **void reset()**

Input:

□ void

Description:

□ resets the internal clock to 0

Example Usage:

□ `stopwatch.reset();`

example usage of the timer class

Demonstration of how to use the timer class to get times of various things

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//make the timer global...as soon as you declare it...it starts recording the time..
timer stopwatch;

int main(int argc,char* argv[]){
    ...put initialization stuff....
    ...
    //resets the timer to 0 so we can time the hard calculation rather then the intialization
    stopwatch.reset();
    ...begin Hard calcualtion...
    ...end Hard caclualtion...

    cout<<"Time Taken: "<<stopwatch()*1e3<<" ms "<<endl; //print out the time...
}
```

--Constructors**--- Random****--- Random****--- Random****--Element Extraction****--- high****--- low****--- offset****--- operator()****--- operator()****--- operator()****--- Random****--- range****--Assignments****--- low****--- offset****--- operator=****--- range**

```
template<class Engine_t=UniformRandom<double> >
class Random : public Engine_t{....
```

This class is the main container for The psuedo-Random number generators...as of now there are only 2..The basic "Random" (generates a uniformly distributed random number) and the "Gaussian Random" generates a gaussian distributed random number.

Engine_t can be...

UniformRandom<T>--> to generate uniformly distributed random numbers

GaussianRandom<T>--> to generate gaussian random numbers

the random number generators use in house functions (not the normal C random functions)

Random::constructor

Function:

Random()

Input:

none

Description:

empty constructor..sets offset=0, and range=1

Example Usage:

Random<> moo;

Random::constructor

Function:

□ `Random(Random &cp)`

Input:

□ `cp-->` another Random of the same type

Description:

□ the copy constructor

Example Usage:

□ `Random<UniformRandom> > mool(-2, 2);`
`Random<UniformRandom> > moo2(mool); //copied`

Random::constructor

Function: □ **Random(NumType low, NumType high)**

Input: □ UniformRandom-->sets lower bound=low and the upper bound=high
 GaussianRandom-->sets offset=low and the varience=high

Description: □ the basic constructor for the random

Example Usage: □ `Random<UniformRandom<> > moo1(-2,2); //lbound=-2, ubound=2`
`Random<GaussianRandom<> > moo2(10,2); //offset=10, varience=2`

Random::element extraction

Function:

NumType high()

Input:

none

Description:

Return the upperbound value

Example Usage:

```
 Random<UniformRandom> > r(-5,5);  
cout<<r.high(); //will print "5"
```

Random::element extraction

Function:

NumType low()

Input:

none

Description:

Return the lowerbound value

Example Usage:

```
 Random<UniformRandom> > r(-5,5);  
cout<<r.low(); //will print "-5"
```

Random::element extraction

Function:

□ **NumType offset()**

Input:

□ none

Description:

□ Return the offset value

Example Usage:

```
□ Random<UniformRandom<> > r(-5,5);  
cout<<r.offset(); //will print "0"
```

Random::element extraction

Function:

□ **NumType operator()**

Input:

□ none

Description:

□ Returns a random number

Example Usage:

□ `Random<UniformRandom<> > r(-4, 5);
double ran=r();`

Random::element extraction

- Function:** `NumType operator(T in, int i)`
- Input:** `in-->Another number....THIS IS A DUMMY!!`
`i--> any integrer...THIS IS A DUMMY`
- Description:** `Return a random number...This operator is used for the "apply" function inside Vector....`
- Example Usage:** `Random<UniformRandom<> > r(-4,5);`
`Vector<double> tv(50);`
`tv.apply(r); //the vector is filled with random numbers`

Random::element extraction

Function: `□ NumType operator(T in, int i, int j)`

Input: `□ in-->Another number...THIS IS A DUMMY!!`
`□ i--> any integer...THIS IS A DUMMY`
`□ j--> Any integer...THIS IS A DUMMY`

Description: `□ Return a random number...This operator is used for the "apply" function inside matrix....`

Example Usage: `□ Random<UniformRandom<> > r(-4,5);`
`□ rmatrix<double> tv(50,50);`
`□ tv.apply(r); //the matrix is filled with random numbers`

Random::element extraction

Function:

□ **NumType Random();**

Input:

□ none

Description:

□ Returns a random number

Example Usage:

□ `Random<UniformRandom<> > r(-4,5);
double ran=r.Random();`

Random::element extraction

Function:

□ **NumType range()**

Input:

□ none

Description:

□ Return the range value

Example Usage:

```
□ Random<UniformRandom<> > r(-5,5);  
cout<<r.range(); //will print "10"
```

Random::assignments

Function: **void low(NumType lo)**

Input: lo--> the new Lower bound

Description: sets the lower bound of the class

Example Usage: `Random<UniformRandom<> > r(-5,5);
r.low(-96); //sets the lower bound to "-96"`

Random::assignments

Function: `void offset(NumType off)`

Input: `off--> the New offset value`

Description: `sets the value of the offset`

Example Usage: `Random<GaussianRandom<> > r(-5,5);
r.offset(6); //sets offset to "6"`

Random::assignments

Function: **Random operator=(Random &rhs)**

Input: rhs--> A Random WITH THE SAME ENGINE!

Description: assigns one Random from another...THe Engine of the "rhs" must be the same as the "lhs"

Example Usage: Random<UniformRandom> myG(5 , 9) ;
 Random<UniformRandom> myG2=myG; //okay assignment
 Random<GaussianRandom> myG3=myG; //WILL FAIL (not compile)

Random::assignments

Function:

□ **void range(NumType ran)**

Input:

□ ran--> the New range value

Description:

□ sets the value for the range...

Example Usage:

□ `Random<UniformRandom<> > r(-5,5);
r.range(0.5); //sets range to 0.5`

--Constructors[--- VNMRstream](#)[--- VNMRstream](#)**--IO**[--- read](#)[--- write](#)**--Other Functions**[--- close](#)[--- is2D](#)[--- is_open](#)[--- open](#)**Examples**[--- file convert](#)

```
class VNMRstream{....
```

This class simply allows the reading a writing of binary VNMR files (the binary FID files produced from Varian's INOVA, or similar system)...you can read and write 1D or 2D data file

IT will correct for any endian-ness (i.e. a SUN vs Linux vs Windows) binary types during the reading process...

VNMR::constructor

Function: **VNMRstream()**

Input: none

Description: empty constructor--> Sets up the default VNMRstream
sets iomode=ios::in | ios::binary

Example Usage: `VNMRstream moo;`
`moo.open("infile"); //opens a file after declaration`

VNMR::constructor

Function: `■ VNMRstream(std::string infile, int iomode=ios::binary | ios::in)`

Input: `■ infile-->the file you wish to begin extracting all the VNMRstream
iomode--> set the input/out mode`

Description: `■ will either open a file for writing or reading....`

Example Usage: `■ VNMRstream moo("myfile.txt", ios::binary | ios::in);`

VNMR::IO

Function: **bool read(Vector<complex> &fid)**
 bool read(matrix &fids)

Input: varname--> the name of the variable within a matlab file...
var--> the variable you wish to fill from the VNMRstream file...the overloads you see above are the types allowed

Description: --fills the 'var' from the data in the VNMRstream file...assuming that 'varname' exists inside the file..
--If the file is a 2D file, you will only get the FIRST fid in the file for the Vector<complex> pieces

Example VNMRstream inmat("fid", ios::binary | ios::in);
Usage: Vector<complex> vdat;
inmat.read(vdat);

VNMR::IO

Function:

- **bool write(Vector<complex> &fid)**
- **bool write(matrix &fids)**

Input:

- var--> the variable you wish to write to the file...

Description:

- puts the data into a VNMR formated file..

Example Usage:

- ```
VNMRstream myout;
myout.open("fid");
Vector<complex> num(1024, 9.0);
myout.write(num);
```

## VNMR::other

**Function:**  **void close()**

**Input:**  none

**Description:**  closes the file...releases the file handle so you can manipulate it elsewhere if you desire

**Example Usage:** 

```
VNMRstream myout;
myout.open("fid");
Vector<complex> num;
myout.read(num);
myout.close();
```

## VNMR::other

**Function:**       **bool is2D( )**

**Input:**       none

**Description:**       checks to see if the file is of 2D format...

**Example Usage:**      

```
VNMRstream myout;
myout.open("fid");
if(myout.is2D()) cout<<"2D vnmr data file"<<endl;
```

## VNMR::other

### Function:

□ **bool is\_open()**

### Input:

□ none

### Description:

□ returns true if the file is open...false if not.

### Example Usage:

```
□ VNMRstream myout;
 myout.open("fid");
 Vector<complex> num;
 myout.read(num);
 if(myout.is_open()) myout.close();
```

## VNMR::other

**Function:**  `void open(const std::string fname, int iomode=ios::binary | ios::in)`

**Input:**  fname--> file name to open...  
iomode --> the mode for the opening...

**Description:**  reads the file into the object

**Example**  `VNMRstream myout;`

**Usage:**  `myout.open("fid");`  
`Vector<complex> num;`  
`myout.read(num);`

## Convert a VNMR file into an acsii or Matlab file

---

This sample program converts a VNMR file into a Matlab 5 of text (ASCII) file

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv){
 int q=1;
 std::string fname="";
 std::string matname;

 //get the VNMR data file name
 query_parameter(argc, argv, q++, "Enter Inova Data file: ", fname);

 //get the out put file name
 query_parameter(argc, argv, q++, "Enter Output file name: ", matname);
 int ch=0;

 //ascii out put or matlab output?
 query_parameter(argc, argv, q++, "Output ASCII or matlab binary[0,1]? ", ch);

 //out storage for reading the fids
 matrix fids;

 //open the VNMR file and read out the fids...
 VNMRstream vfile(fname,ios::binary | ios::in);
 vfile.read(fids);

 //output matlab if desired
 if(ch==1){
 std::string vname=matname;
 if(matname.find(".mat")>matname.size()){
 matname+=".mat";
 }else{
 vname=matname.substr(0, matname.find(".mat"));
 }
 matstream matout(matname.c_str(), ios::binary | ios::out);
 matout.put(vname, fids);
 matout.close();
 std::cout<<" Data saved in Matlab file "
 <<matname<<" under variable '"<<vname<<'"<<std::endl;
 //or out put ASCII
 }else{
 ofstream oo(matname.c_str());
 if(!vfile.is2D()){
 for(int i=0;i<fids.rows();++i){
 oo<<fids(i,0).Re()<<" "<<fids(i,0).Im()<<std::endl;
 }
 }else{
 for(int i=0;i<fids.rows();++i){
 for(int j=0;j<fids.cols();++j){
 oo<<i<<" "<<j<<" "<<fids(i,j).Re()<<" "<<fids(i,j).Im()<<std::endl;
 }
 }
 }
 }
}
```

```
 }
}
std::cout<<" Data saved in ASCII file "<<matname<<std::endl;
}
}
```

---

**--Public Vars**

--- interactive  
--- paramsep  
--- secclose  
--- secopen

**--Constructors**

--- Parameters  
--- Parameters  
--- Parameters  
--- Parameters  
--- Parameters  
--- Parameters

**--Element Extraction**

--- getParamC  
--- getParamCoordD  
--- getParamCoordI  
--- getParamD  
--- getParamI  
--- getParamS  
--- getParamVectorD  
--- getParamVectorI  
--- section

**--Assignments**

--- operator=

--- operator=

--- setParam

**--IO**

--- operator<<

--- print

--- read

**--Other Functions**

--- addSection

--- empty

**Examples**

--- 3 sections

class Parameters{ ....

---

This class makes inputting parameters into programs much simpler...the typical method for inputting parameters is the simple console-ask console-get...but for sufficiently complex programs there could be many different parameters and typeing them all into the cosole everytime can be quite painfull. This class will take in files and parse them accroding to 'sections' and you can then 'get' parameters from those sections easily...

---

## Parameters::Public Vars

**Function:**  **bool interactive**

Input:

Description:  if "true" asks you for any missing parameter you ask for, if "false" returns a '0' of the type

Example Usage:

## Parameters::Public Vars

**Function:**       **char paramsep**

**Input:**           

**Description:**         the separator for a given parameter (the default is '' or a white space)

**Example Usage:**

## Parameters::Public Vars

**Function:**  **string secclose**

**Input:**

**Description:**  the closeing string for a main section (default = "}")

**Example Usage:**

## Parameters::Public Vars

**Function:**  **string secopen**

**Input:**

**Description:**  the opening string for a main section (default = "{}")

**Example Usage:**

## Parameters::constructor

### Function:

**Parameters()**

### Input:

none

### Description:

empty constructor--> Sets up the default Parameters  
secopen="{"  
secclose="}"  
paramsep=' ' (white space)  
interactive=true

### Example Usage:

`Parameters myP;`

## Parameters::constructor

**Function:**  `Parameters(std::string infile, string seco="", string secc="", char params=' ', bool interacti=true)`

**Input:**  infile-->the file you wish to begin extracting all the parameters  
the rest are the parameters default setup

**Description:**  creates an Parameters data set from the file named 'infile'

**Example**  `Parameters moo( "myfile.txt" );`

**Usage:**

## Parameters::constructor

**Function:**  `Parameters(std::ifstream infile, string seco="", string secc="", char params='', bool interacti=true)`

**Input:**  infile-->the file you wish to begin extracting all the parameters...this assumes an already open file..  
the rest are the parameters default setup

**Description:**  creates an Parameters data set from the in stream 'infile'

**Example**  `ifstream myfile("input.txt");`

**Usage:**  `Parameters pset(myfile);`

## Parameters::constructor

**Function:** `Parameters(const Parameters &cp)`

**Input:** `cp-->` a parameter you wish to copy

**Description:** `The copy constructor`

**Example Usage:**

```
Parameters myP("moo.txt");
Parameters myCp(myP);
```

## Parameters::constructor

**Function:**  `Parameters(const Vector<std::string> &in, std::string open="{}", std::string cl="}", char psep=' ', bool interac=true);`

**Input:**  in-->the Vector<string> you wish to begin extracting all the parameters...this assumes an already open file..  
the rest are the parameters default setup

**Description:**  initializes a Parameter object assuming that the Vector<string> if the entire file

**Example**  `Vector<std::string> myS;`

**Usage:**  `myS.push_back("moo 45");  
Parameters myP(myS);`

## Parameters::element extraction

**Function:**  `char getParamC(string::string param, std::string sec="", bool interactive=true, char default=' ')`

**Input:**  Returns a character from the section 'sec' and a parameter names 'param.' If 'sec'==" " then it will look for the first parameter by that name it finds (in ANY section)  
You must 'add' the section inorder to grab anything from that section.

**Description:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
inertactive--> if true, it will Force this parameter to be present (if it is not present it willprompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Example**  `Parameters pset("input.txt");`

**Usage:**  `pset.addSection("asecname");  
char mypar=pset.getParamC("par", "asecname");`

## Parameters::element extraction

**Function:**  `coord<> getParamCoordD(string::string param, std::string sec="", char csep=',', bool interactive=true, coord<> default=(0.0,0.0,0.0))`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
csep--> how individual elements are separated in the list  
inertactive--> if true, it will Force this parameter to be present (if it is not present it willprompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a coord<> from the section 'sec' and a parameter names 'param.' If 'sec'=="" then it will look for the first parameter by that name it finds (in ANY section)

the default format is

varname 3,5,7

the individual values are separated by COMMAS

you must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset( "input.txt" );  
pset.addSection( "asecname" );  
coord<> mypar=pset.getParamCoordD( "varname" , "asecname" );`

**Usage:**

## Parameters::element extraction

**Function:**  `coord<int> getParamCoordI(string::string param, std::string sec="", char csep=',', bool interactive=true, coord<int> default=(0,0,0))`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
csep--> how individual elements are separated in the list  
interactive--> if true, it will Force this parameter to be present (if it is not present it willprompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a coord<> from the section 'sec' and a parameter names 'param.' If 'sec'==" then it will look for the first parameter by that name it finds (in ANY section)

the default format is

varname 3,5,7

the individual values are separated by COMMAS

you must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset("input.txt");  
pset.addSection("asecname");  
coord<int> mypar=pset.getParamCoordI("varname", "asecname");`

**Usage:**

## Parameters::element extraction

**Function:**  `double getParamD(string::string param, std::string sec="", bool interactive=true, double default=0.0)`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
interactive--> if true, it will Force this parameter to be present (if it is not present it will prompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a double from the section 'sec' and a parameter names 'param.' If 'sec'=="" then it will look for the first parameter by that name it finds (in ANY section)  
you must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset( "input.txt" );`

**Usage:**  `pset.addSection( "asecname" );  
double mypar=pset.getParamD( "par" , "asecname" );`

## Parameters::element extraction

**Function:**  `int getParamI(string::string param, std::string sec="", bool interactive=true, int default=0)`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
interactive--> if true, it will Force this parameter to be present (if it is not present it will prompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns an Integer from the section 'sec' and a parameter names 'param.' If 'sec'=="" then it will look for the first parameter by that name it finds (in ANY section)  
You must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset( "input.txt" );`

**Usage:**  `pset.addSection( "asecname" );  
int mypar=pset.getParamI( "par" , "asecname" );`

## Parameters::element extraction

**Function:**  `std::string getParamS(string::string param, std::string sec="", bool interactive=true, string default "")`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
interactive--> if true, it will Force this parameter to be present (if it is not present it will prompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a String from the section 'sec' and a parameter names 'param.' If 'sec'=="" then it will look for the first parameter by that name it finds (in ANY section)  
You must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset("input.txt");`

**Usage:**  `pset.addSection("asecname");  
std::string mypar=pset.getParamS("par", "asecname");`

## Parameters::element extraction

**Function:**  `Vector<double> getParamVectorD(string::string param, std::string sec="", char csep=',', bool interactive=true, Vector<> default=0)`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
csep--> how individual elements are separated in the list  
interactive--> if true, it will Force this parameter to be present (if it is not present it willprompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a `Vector<double>` from the section 'sec' and a parameter names 'param.' If 'sec'==" then it will look for the first parameter by that name it finds (in ANY section)

the default format is

varname 3,5,7,8.0,...

the individual values are separated by COMMAS

you must 'add' the section inorder to grab anything from that section.

**Example**

```
 Parameters pset("input.txt");
pset.addSection("asecname");
Vector<double> mypar=pset.getParamVectorD("varname", "asecname");
```

**Usage:**

## Parameters::element extraction

**Function:**  `Vector<int> getParamVectorI(string::string param, std::string sec="", char csep=',', bool interactive=true, Vector<int> default=0)`

**Input:**  param-->the parameter name you wish to get..  
sec--> the section you wish to grab the parameter from  
csep--> how individual elements are separated in the list  
inertactive--> if true, it will Force this parameter to be present (if it is not present it willprompt you for it), if false, it will return the default value if the parameter is not found in the file  
default--> the default value to return IF interactive=false and the parameter is not present in the parameter file

**Description:**  Returns a `Vector<int>` from the section 'sec' and a parameter names 'param.' If 'sec'==" then it will look for the first parameter by that name it finds (in ANY section)

the default format is

varname 3,5,7,8,...

the individual values are separated by COMMAS

you must 'add' the section inorder to grab anything from that section.

**Example**  `Parameters pset( "input.txt" );  
pset.addSection( "asecname" );  
Vector<int> mypar=pset.getParamVectorI( "varname" , "asecname" );`

**Usage:**

## Parameters::element extraction

**Function:**  `Vector<std::string> section(std::string secname)`

**Input:**  secname--> the name of the section you wish to obtain...

**Description:**  Returns the file section (line by line) from the section name 'secname' YOU MUST have used 'addSection(secname)' first inorder to grab the section (else it will be a NULL vector)

**Example**  `Parameters pset( "input.txt" );`

**Usage:**  `pset.addSection( "asecname" );`

`Vector<std::string> thesec=pset.section( "asecname" );`

## Parameters::assignments

**Function:** `Parameters &operator=(const Parameters &rhs);`

**Input:** `rhs->` a parameter you wish to copy

**Description:** `assigns one Parameters from another.`

**Example Usage:**

```
ifstream myfile("input.txt");
Parameters pset(myfile);
Parameters pset2;
pset2=pset;
```

## Parameters::assignments

**Function:**  **Parameters &operator=(const std::ifstream &file);**

**Input:**  file--> an input file stream

**Description:**  assigns the Parameters from input file stream

**Example Usage:**  `ifstream myfile( "input.txt" );`  
`Parameters pset=myfile;`

## Parameters::assignments

**Function:**

- `void setParam(std::string par, int toset, std::string sec="");`
- `void setParam(std::string par, double toset, std::string sec="");`
- `void setParam(std::string par, std::string toset, std::string sec="");`
- `void setParam(std::string par, char toset, std::string sec="");`
- `void setParam(std::string par, Vector<double> toset, std::string sec="");`
- `void setParam(std::string par, Vector<int> toset, std::string sec="");`
  
- `template<int N>`
- `void setParam(std::string par, coord<double, N> toset, std::string sec="")`
- `template<int N>`
- `void setParam(std::string par, coord<int,N> toset, std::string sec="")`

**Input:**

- `par`--> the parameter variable name
- `toset`--> the value the 'par' should have
- `sec`--> which section this parameters lives in

**Description:**

- sets the parameter with the name 'par' to the value 'toset'. If the variable name is not present it will create it.

**Example**

- `Parameters myP( "input.txt" );`

**Usage:**

- `myP.setParam( "myvar" , 5.0 );`

## Parameters::IO

**Function:**  `std::ostream &operator<<(std::ostream &oo, Parameters &out)`

**Input:**  `oo-->` an output stream

`out-->` the parameters you wish to write

**Description:**  prints the Parameters out to the ostream

**Example Usage:**  `Parameters myP("input.txt");`

`myP.setParam("varname", 77);`

`cout<<myP;`

## Parameters::IO

**Function:**

- `bool print(std::ostream &out);`
- `bool print(std::string out);`
- `bool print(const char *out);`

**Input:**

- `out-->` the file name, or the ostream to print the parameter set to

**Description:**

- writes the parameters to a file or to an ostream. if something goes wrong it will return false

**Example Usage:**

- `Parameters myP( "input.txt" );`
- `myP.setParam( "loo", 5 );`
- `myP.print( "outpar.txt" );`

## Parameters::IO

### Function:

- `void read(const std::string fname)`
- `void read(std::ifstream &file)`

### Input:

- `fname`--> file name to open...passes it to 'read(ifstream)'
- `file`--> an already opened ifstream...

### Description:

- reads the file into the object

### Example Usage:

```
□ Parameters pset;
 pset.read("input.txt");
 ifstream file("input.txt");
 pset.read(file);
```



## Parameters::other

|                       |                                                                                              |
|-----------------------|----------------------------------------------------------------------------------------------|
| <b>Function:</b>      | <input type="checkbox"/> <b>bool empty()</b>                                                 |
| <b>Input:</b>         | <input type="checkbox"/> <b>void</b>                                                         |
| <b>Description:</b>   | <input type="checkbox"/> if there is nothing in the file this is true                        |
| <b>Example Usage:</b> | <input type="checkbox"/> <code>Parameters myP;</code><br><code>myP.empty(); //is true</code> |

parses 3 sections in an input file

---

Assume that you have a file input that is composed of 3 main sections...a 'spin' section, 'params' section, and a 'pulse' section (a typical NMR type experiment) and that this file looks like this

```
-----input.txt-----
#comments are easily made by useing the '#'

#we can define global variables...
napps 23

#this 'spin' input section is the exact type taken by the class 'SolidSys'
spin{
 numspin 2
 T 1H 0
 T 13C 1
 C 1200 455 0 1
 C 0 3400 0 0
 D 2000 0 1
}
params{
 wr 2000
 npts 512
#this is a LOCAL variable (local to 'params')
#it does not overwrite any global ones...
 napps 12
 fileout data
}
pulse{
 angle 90
 phase 0
}
-----input.txt-----

//NOW for the source code to read in and extract things from this parameter file

#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv){

//get the file name to parse
 std::string fname;
 query_parameter(argv, argc, 1, "Enter File Name to parse:", fname);

//Set up the parameters object
 Parameters pset(fname);

//need to tell the parameters what sections are in the file
```

```
pset.addSection("spin");
pset.addSection("params");
pset.addSection("pulse");

//The spins section is already handled by the 'SolidSys' class
//so we pass that entire section to it
SolidSys mysys(pset.section("spin"));

//now to grab the other parts
//gets a 'double' called 'wr' from the section 'param'
double wr=pset.getParamD("wr", "params");
//gets an 'integer' calls 'napps' from the section 'param'
//if the parameter is not preset it will not ask for it, instead it will
// return its default value of '90'
int napps=pset.getParamI("napps", "params", false, 90);

//this grabs out 'global variable'
int napGlobal=pset.getParamI("napps");

//get the pulse pieces
double ang=pset.getParamD("angle", "pulse");
double phase=pset.getParamD("phase", "pulse");

//get the the 'file out' string
std::string fout=pset.getParamS("fileout", "params");

//now 'do things' with all you inputted dat.....
}
```

---

--Global Constants class Parser { ...

--- ParserGlobalVars

--Constructors

--- Parser

--- Parser

--Assignments

--- operator=

--Other Functions

--- addGlobalVar

--- addVar

--- getVar

--- operator()

--- parse

Examples

--- simple Parseing

The parser class acts as a simple math expression parser for an input string.  
The available math functions included so far are

exp, ln, log, sin, cos, tan, asin, acos,  
atan, sinh, cosh, tanh, asinh, acosh,  
atanh, abs, floor, ceil, round, sqrt,  
int

and the math operators

- , + , / , \* , ^ , == , != , <= , >= , < , > , && ,  
||

and these GLOBAL constants

e, pi, deg2rad, rad2deg

It can handle levels (i.e. parenthesis) so a general expression input string would look something like

"2\*4+cos(5)^(-1)-ln(78)"

The only two functions you need to know about are 'parse' and 'operator()'. The parse function parses the input into a stack, the 'operator()' (or 'evaluate()') returns the double from the expression stack.

Variables can be used as well. There are two levels of variable input, global and local. Global variables are placed into a container called 'ParserGlobalVars' and are visible to ALL instances of 'Parser.' To add your own variable, simply add it (using the 'addVar' function). It will first look in the local variable set and then in the Global variable set (so you can have local and global variables of the same name, it will use the local one first).

## Parser::global constant

**Function:**  `std::map<std::string, double> ParserGlobalVars;`

**Input:**

**Description:**  This is the main container for the Global variables.

**Example Usage:**

## Parser::constructor

### Function:

**Parser()**

### Input:

void

### Description:

The empty constructor. It makes sure the global constants are set.

### Example Usage:

`//a simple object  
Parser myParse;`

## Parser::constructor

**Function:**       **Parser(std::string inExpr)**

**Input:**       inExpr--> a valid string expression

**Description:**       Constructs the object, sets the global variables, and then parsed the input string.

**Example Usage:**       std::string myE="45+99" ;  
                                Parser myParse(myE) ;

## Parser::assignments

### Function:

□ **void operator=(const Parser &rhs);**

### Input:

□ rhs--> a Parser object

### Description:

□ Simply copies all the data in the Parser object to another.

### Example Usage:

□ `Parser myParse( "5^(-1)" );`

```
Parser myP2;
myP2=myParse;
```

## Parser::other

**Function:** `void addGlobalVar(std::string name, double value);`

**Input:** `name--> the variable string name`  
`value--> the new value for the variable`

**Description:** `This function will add a variable to the GLOBAL variable set if it is not present. If the variable exists already it will simply replace the value.`

**Example**    `Parser myParse;`

**Usage:**    `//add a var`

```
myParse.addGlobalVar("s", 3);
myParse.parse("23*s");

//these 2 do the same thing
//should print '69 69'
cout<<myParse.evaluate()<< " "<<myParse()<<endl;

//This adds a LOCAL variable 'r'
myParse.addVar("r", 34);

//reevaluate the expression
// this will print '782 782'
cout<<myParse.evaluate()<< " "<<myParse()<<endl;

//create another object
Parser myParse2;
//uses the global 's' variable
myParse2.parse("s+4");

//this will print '7 7'
cout<<myParse.evaluate()<< " "<<myParse()<<endl;
```

## Parser::other

**Function:** `void addVar(std::string name, double value);`

**Input:** `name`--> the variable string name  
`value`--> the new value for the variable

**Description:** `This function will add a variable to the LOCAL variable set if it is not present in EITHER the local or global variable set. If the variable exists it will simply replace the value. If the variable exists in the global set, it will replace the value there.`

**Example**    `Parser myParse;`

**Usage:**    `//add a var`

```
myParse.addVar("s", 3);
```

```
myParse.parse("23*s");
```

```
//these 2 do the same thing
```

```
cout<<myParse.evaluate()<<" "<<myParse()<<endl;
```

```
//update the variable 's'
```

```
myParse.addVar("s", 34);
```

```
//reevaluate the expression
```

```
cout<<myParse.evaluate()<<" "<<myParse()<<endl;
```

## Parser::other

**Function:** `double getVar(std::string name);`

**Input:** `name`--> the variable name

**Description:** `This returns the value of the variable (if present). It will return 0 and print a warning if the variable is NOT present.`

**Example** `Parser myParse;`

```
//add a var
myParse.addVar("s", 3);
//prints '3'
cout<<myParse.getVar("s");
```

**Usage:**

## Parser::other

**Function:**  **double operator()();**  
 **double evalutate();**

**Input:**  void

**Description:**  This evaluates the expression which 'parse' parsed up into the stack. At this point the variables are expressed if any are present.

**Example**  Parser myParse;

**Usage:** myParse.addVar("s", 3);  
myParse.parse("23\*s");

```
//these 2 do the same thing
cout<<myParse.evaluate()<<" "<<myParse()<<endl;
```

## Parser::other

**Function:**  `bool parse(std::string inExpr);`  
 `bool parse(const char *inExpr);`

**Input:**  `inExpr--> a valid input math expression string`

**Description:**  Parses an expression. It does not evaluate any variables until 'operator()' is called, it simply creates the stack operations. If the parse operation fails it returns false, otherwise it returns true.

**Example**  `Parser myP;`

**Usage:**  `myP.parse( "34*cos(pi)");`

A simple 'HOW-TO' Parser example.

---

This example simply runs through the various functions and expression parsing available to you.

---

```
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

int main(){

 //declare the object
 Parser myParse;

 //add a global variable
 myParse.addGlobalVar("m", 98);

 //add a local variable
 myParse.addVar("g", 3);

 //the addVar command will replace a variables value if
 //it has been inserted already. It will first look to the local set
 //if it is not found there it will try the global set

 //this will replace the global variable 'm'
 myParse.addVar("m", 2);
 //now we can parse some expression
 myParse.parse("m*6+pi+g");

 //get the value from the expression
 //this should print '18.1456'
 cout<<myParse()<<endl;

 //we can declare another Parser object that will be able to see
 // the global 'm' variable
 Parser myParse2;

 //parse an expression
 myParse2.parse("m*(3+5)");

 //get the value
 //should print out '16'
 cout<<myParse2()<<endl;
}
```

**--Public Vars**

--- myParse

**--Constructors**

--- ScriptParse

--- ScriptParse

--- ScriptParse

**--Other Functions**

--- decide

--- parse

--- parse

**Examples**

--- 1) using this class

--- 2) decide

```
class ScriptParse { ...
```

---

This takes in a Vector as the input that allows for variable setting, simple looping, and simple if, elseif, else structures...

It is designed to be a base class for specific 'functional' classes where input scripts could be necessary....sub-classes only need to create another virtual function called 'decide' to add additional functional input.

It determines what to do if it hits a 'loop', an assignment (A=B), or 'if/else if/else' statements. Currently there are not 'break' or 'return' keywords

Assignments can be made using simple syntax

A=B (A gets set from B)

where B can be some expression  
loops are performed via the

```
loop(i=1:40)
 ...
 end
```

syntax where the i=1:40 means "loop from 1 to 40" and

```
if(t==9)
 ...
 else if(moo==34)
 ...
 else
 ...
end
```

It also has a print function as well that will display the argument to the terminal

```
print(56)
```

---

## ScriptParse::Public Vars

**Function:**  **Parser myParse;**

Input:

Description:  This an instance of the Parser object that handles all the expression evaluation. The ScriptParse add variables to the LOCAL (unless they are already declared in the global set) variable set in 'myParse'

Example

Usage:

## ScriptParse::constructor

### Function:

- ScriptParse()**

Input:

- void

Description:

- The empty constructor.

Example Usage:

- `ScriptParse myScr;`

## ScriptParse::constructor

**Function:** `ScriptParse(Parameters &pset);`

**Input:** `pset-->` a parameter set with the main section already added

**Description:** `Uses the main parameter set section ('section("")') to use as the script input. It will immediately parse in the input string.`

**Example** `std::string inputFile="someName";`

**Usage:** `Parameter pset(inputFile);`

```
//this will 'run' in the script
ScriptParse myScr(pset);
```

## ScriptParse::constructor

**Function:**  `ScriptParse(const Vector<std::string> &pset);`

**Input:**  `pset-->` a `Vector<std::string>` containing the script to parse.

**Description:**  Uses the string list as the script input. It will immediately parse in the input string.

**Example Usage:**  `Vector<std::string> myP;  
//..fill myP with something...  
  
//this will run the input  
ScriptParse myScr(myP);`

## ScriptParse::other

**Function:**  **virtual bool decide(std::string inSqe);**

**Input:**  inSqe--> a single line from the input script

**Description:**  This is the main virtual function that sub class should rewrite, but the sub classes should also point to this function as the last step(i.e. ScriptParse::decide(inSqe)) as loops and ifs and variables parsing are still needed. This function decides what to do with the input string (see the example for more details).

**Example**  (see 'decide' example)

**Usage:**

## ScriptParse::other

**Function:** `bool parse()`

**Input:** `void`

**Description:** `Will re-run a script if the data vector has already been inputted. This is so you can run the script as many times as you wish. It returns false if a parse error occurred.`

**Example** `std::string inputFile="someName";`

**Usage:** `Parameter pset(inputFile);`

```
//this will 'run' in the script
ScriptParse myScr(pset);
```

```
//re run the script again
myScr.parse();
```

## ScriptParse::other

**Function:**  `bool parse(Parameter &pset);`  
 `bool parse(const Vector<std::string> &pset);`

**Input:**  pset--> a valid parameter object or a list of strings

**Description:**  Will run a script give the input data vector (or the pset.section("")). This will reparse the script and run it. If something fails it will return false, otherwise it returns true.

**Example**  `std::string inputFile="someName";`

**Usage:**  `Parameter pset(inputFile);`

```
//this sets up an object
ScriptParse myScr;
```

```
//run the script again
myScr.parse(pset);
```

## Using the ScriptParse class for simple print operations

---

This example is quite basic, and does not do much because this class is meant to be a superclass to other objects.

---

```
-----the input file-----
myscript{
 B=2
 a=B

 loop(i=5:10)
 print(a*i)
 a=a+1
 if(a==4)
 print(77)
 end
 end

}

-----end Input file-----
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv){

 std::string fname;
 query_parameter(argc, argv, 1, "Input File Name: ", fname);
 Parameters pset(fname);

 ScriptParse myP;
 myP.parse(pset);
}

//----This should print out-----
//a*i=10
//a*i=18
//a*i=28
//77=77
//a*i=40
//a*i=54
//a*i=70
```

---

How to extend this class to use for your own special functions

---

This example creates a class based on ScriptParse that will perform more tasks given an input script.

---

```
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

// As simple on how-to extend the ScriptParse class

class myScriptClass:
 public ScriptParse
{
 private:
 //these functions are the actuall 'dooers' of
 //the script when the script hits these things

 //this performs a more complex print operation with
 // the input...uses the 'myPrint' keyword
 void domyPrint(std::string inSqe);

 //this dumps a matrix out to the screen
 // uses the key word 'makematrix'
 void doMakeMatrix(std::string inSqe);

 public:
 //wrap the basic constructos back to the
 //script parse class
 myScriptClass():
 ScriptParse()
 {}
 myScriptClass(Parameters &pset):
 ScriptParse(pset)
 {}
 myScriptClass(const Vector<std::string> &in):
 ScriptParse()
 {}

 //this is the new decide function
 bool decide(std::string inSqe);
};

//create the functions

//first create our new decide function
bool myScriptClass::decide(std::string inSqe)
{
```

```

if(inSqe.find("makematrix(")<inSqe.size())){
 doMakeMatrix(inSqe);
}else if(inSqe.find("myPrint(")<inSqe.size()){
 domyPrint(inSqe);
}else{
 //wrap any other input back to the
 // base class so that loop and assignments
 //and if's work as well
 return ScriptParse::decide(inSqe);
}
return true;
}

//this parse the 'myPrint' statement
// simply prints out what is given to it
//(DOES NOT evaluate the expression)
//the syntax:: myprint(monkey)
// will print 'monkey' to cout
void myScriptClass::domyPrint(std::string inSqe)
{
 if(inSqe.find("myPrint(")<inSqe.size())
 {
 std::string tmS=getInside(inSqe); //get the stuff inside the '()' '
 if(tmS.size()==0)
 {
 std::cerr<<"error: myScriptClass::domyPrint"<<std::endl;
 std::cerr<<" Bad 'on' usage for ""<<inSqe<<""<<std::endl;
 std::cerr<<" should be \"myprint(thing)\" "<<endl;
 throw BL_exception(__FILE__, __LINE__);
 }
 //just dump the input to the screen
 std::cout<<tmS<<endl;
 }
}

//this parse the 'makematrix' statement
// creates a real matrix, then prints it to the screen
//the syntax:: makematrix(3,3)
// creates a 3x3 real matrix filled with 0's
//the syntax:: makematrix(3,3,5)
// creates a 3x3 real matrix filled with 5's
void myScriptClass::doMakeMatrix(std::string inSqe)
{
 Vector<std::string> ps;
 //remove any white spaces in the input
 std::string tmS=removeWhite(inSqe);
 if(tmS.find("makematrix(")<tmS.size())
 {
 tmS=getInside(tmS); //get the stuff inside the '()' '
 ps=parseComma(tmS); //split the commas in the function
 if(ps.size()<=1)
 {

```

```

 std::cerr<<"error: myScriptClass::doMakeMatrix"<<std::endl;
 std::cerr<<" Bad 'on' usage for ""<<inSqe<<""<<std::endl;
 std::cerr<<" should be \"makematrix(rows, cols)\" "<<endl;
 std::cerr<<" or \"makematrix(rows, cols, filler)\" "<<endl;
 throw BL_exception(__FILE__, __LINE__);
 }
//our matrix object
rmatrix mymat;
//the input parameters could be expression
//so we need to parse them up using the 'myParse' object
myParse.parse(ps[0]); //the rows
int rows=int(myParse());
myParse.parse(ps[1]); //the cols
int cols=int(myParse());

double filler=0;
//get the filler if any
if(ps.size()>2){
 myParse.parse(ps[2]);
 filler=myParse();
}

//resize the matrix
mymat.resize(rows, cols, filler);
//just dump the matrix to the screen
std::cout<<mymat;
}
}

/*
#---- an example input file ----

A=2;
loop(i=1:3)
 myPrint(on the matrix tonight)
 print(A)
 if(i==2)
 makematrix(i,A+3, 5)
 else
 makematrix(i,A)
 end
end
*/

```

#-----end example input file-----

----- THE EXPCTED OUTPUT  
Here is the output

-----
on the matrix tonight
A=2
Full matrix: 1x2
[ [ 0 0 ] ]

```
on the matrix tonight
A=2
Full matrix: 2x5
[[5 5 5 5 5]
[5 5 5 5 5]
]
on the matrix tonight
A=2
Full matrix: 3x2
[[0 0]
[0 0]
[0 0]
]

*/
int main(int argc, char **argv)
{
 std::string fname;
 query_parameter(argc, argv, 1, "Input File Name: ", fname);

 Parameters pset(fname);

 myScriptClass myScr;
 myScr.parse(pset);
}
```

---

**--Public Vars**[--- acq1D](#)[--- acq2D](#)**--Constructors**[--- XWINNMRstream](#)[--- XWINNMRstream](#)**--IO**[--- read](#)**--Other Functions**[--- close](#)[--- is2D](#)[--- is\\_open\(\)](#)[--- open](#)**Examples**[--- converter](#)

class XWINNMRstream{ ...

---

This is quite a simple class in its function, when given a Directory name, it will determine the necessary parameters to read the binary data contained in the 'fid' or 'ser' files into a matrix (2D) or vector (1D). It also gives you access to the elements inside the 'acqu' and 'acqu2' files inside the directory.

The XWINNMR data directory is organized like

expname/

fid--> the binary data of 1Ds

ser--> the binary data of 2Ds

acqu --> the 1D parameters

acqu2 --> the 2D auxillary stuff

... --> other things i do need to know about  
to read the files

Currently it does not write anything because that would require me to know everything about that entire directory structure and the files in it, which i do not....

---

## XWINNMR::Public Vars

**Function:**             **XWINNMRacqu acq1D;**

**Input:**           

**Description:**         This is the parameter file parser for the 1D file 'acqu'.

**Example Usage:**      
                        XWINNMRstream moo( "mydata/" );  
                        double sw=0;  
                        //get the sweep width in Hz  
                        moo.acq1D.get( "SW\_h" , sw );  
                        cout<<sw<<endl;

## XWINNMR::Public Vars

**Function:**       **XWINNMRacqu acq2D;**

**Input:**           

**Description:**         The is the parser object for the 2D parameter file 'acqu2'.

**Example Usage:**      
                        XWINNMRstream moo( "mydata/" );  
                        int td=0;  
                        //get the number of pts in the  
                        // second dimension  
                        moo.acq2D.get( "TD" , td );  
                        cout<<td<<endl;

## XWINNMR::constructor

**Function:**  **XWINNMRstream( )**

**Input:**  void

**Description:**  Does nothing...to use the class you must use 'open'

**Example Usage:**  XWINNMRstream moo;  
moo.open( "myDir/" );

## XWINNMR::constructor

**Function:** `□ XWINNMRstream(std::string dir);`

**Input:** `□ dir--> the directory of the data set`

**Description:** `□ This opens up the 'acqu' and 'acqu2' (if present) and the binary data file...`

**Example Usage:** `□ XWINNMRstream moo( "mydir/" );`

## XWINNMR::IO

**Function:**  `bool read(Vector<complex> &oneD);`  
 `bool read(matrix &twoD);`

**Input:**  oneD--> a complex vector to be filled with 1D data  
twoD--> a complex matrix to be filled with 2D data

**Description:**  This will read the 1D or 2D data from the binary file, converting the endian's as nessessary into the input containers.

returns 'true' if it worked.

**Example**  `XWINNMRstream moo( "mydata/" );`

**Usage:** `matrix td;`

```
Vector<complex> od;
if(moo.is2D()) moo.read(td);
else moo.read(od);
```

## XWINNMR::other

### Function:

□ **void close()**

### Input:

□ void

### Description:

□ Closes all the files and releases all file handels.

### Example Usage:

```
□ XWINNMRstream moo("mydata/");
 //this is a boring example
moo.close();
```

## XWINNMR::other

### Function:

□ **bool is2D()**

### Input:

□ **void**

### Description:

□ Returns 'true' if the input directory contains 2D data...

### Example Usage:

```
□ XWINNMRstream moo("mydata/");
 if(moo.is2D()){
 cout<<"Yea!! a 2D set..."<<endl;
 }
```

## XWINNMR::other

**Function:**             **bool is\_open()**

**Input:**             **void**

**Description:**         Have we opened up the binary file for reading yet?

**Example Usage:**     `XWINNMRstream moo( "mydata/" );  
if(moo.is_open()){  
cout<<"okay i'm open"<<endl;  
}`

## XWINNMR::other

**Function:**  **bool open(std::string dir)**

**Input:**  dir--> a directory name

**Description:**  This will open up all the nessesaray file for reading: the 'acqu', the 'acqu2' (if present) and the binary data file (a 'fid' or 'ser').

returns 'true' if everything worked

**Example**  XWINNMRstream moo;

**Usage:**  moo.open( "mydata/ " );

---

A simple program to convert XWINNMR data to Matlab or Text Data files

```
#include "blochlib.h"
//the required 2 namespaces
using namespace BlochLib;
using namespace std;

// Converts binary Bruker Xwin NMR data (BIG ENDIAN)
// to a matlab file for a 1D spectrum

int main(int argc, char **argv)
{
 int q=1;
 std::string fname="", matname;
 int choppts;
 std::cout<<std::endl<<"**BlochLib XWINNMR (Bruker) 'directory' --> Matlab converter"<<std::endl;
 query_parameter(argc, argv, q++, " --Enter Burker Data Directory: ", fname);
 query_parameter(argc, argv, q++, " --Enter Output File Name: ", matname);
 int txt=0;
 query_parameter(argc, argv, q++, " --Text[0] or Matlab[1] ", txt);

 std::string Brmmessage=" --Bruker tends to add ~70 pts of bad ";
 Brmmessage+=" 'Digitizer garbage' at the begining of ";
 Brmmessage+=" each FID, should i remove these points? ";
 Brmmessage+=" -if so how Many pts to remove? (enter '0' for none):";

 query_parameter(argc, argv, q++,Brmmessage, choppts);

 XWINNMRstream loadF(fname);
 double sw, sw2;
 int TD, TD2;
 std::string nnuu;

 loadF.acq1D.get("SW_h", sw);
 cout<<" *Sweep width in 1D: "<<sw<<"Hz"<<endl;

 loadF.acq1D.get("TD", TD);
 cout<<" *points in 1D: "<<TD/2<<endl;

 if(loadF.is2D()){
 loadF.acq2D.get("SW_h", sw2);
 cout<<" *Sweep width in 2D: "<<sw2<<"Hz"<<endl;

 loadF.acq2D.get("TD", TD2);
 cout<<" *points in 2D: "<<TD2<<endl;
 }

 matsstream mm;
 std::string vname=matname;

 if(txt==1){
 if(matname.find(".mat")>matname.size()){
 matname+=".mat";
 }else{
 }
```

```

 vname=matname.substr(0, matname.find(".mat"));
}
mm.open(matname, ios::out | ios::binary);
}

matrix fid2d;
Vector<complex> fid;
if(!loadF.is2D()){
 loadF.read(fid);
 if(choppts>0)
 { fid=fid(Range(choppts, fid.size()-1)); }

 if(txt==1){
 mm.put(vname, fid);
 mm.put("sw1", sw);
 std::cout<<" *Data Saved in '"<<matname
 <<"' under the variable '"<<vname<<"' "<<std::endl;
 std::cout<<" *The Sweep Width information is also stored as variables"
 <<" 'sw1' for 1D info"<<std::endl;
 }else{
 std::ofstream of(matname.c_str());
 for(int i=0;i<fid.size();++i){
 of<<fid[i].Re()<<" "<<fid[i].Im()<<std::endl;
 }
 std::cout<<" *Data Saved in '"<<matname
 <<"' (in cols <real> <imag>)"<<std::endl;
 }
}else{
 loadF.read(fid2d);
 matrix tmFid=fid2d;
 if(choppts>0)
 { tmFid=fid2d(Range(0, fid2d.rows()-1),Range(choppts, fid2d.cols()-1)) ; }

 if(txt==1){
 mm.put(vname, tmFid);
 mm.put("sw1", sw);
 mm.put("sw2", sw2);
 std::cout<<" *Data Saved in '"<<matname
 <<"' under the variable '"<<vname<<"' "<<std::endl;
 std::cout<<" *The Sweep Width information is also stored as variables"
 <<" 'sw1' for 1D info"
 <<" 'sw2' for 2D info"<<std::endl;
 }else{
 std::ofstream of(matname.c_str());
 for(int i=0;i<fid2d.rows();++i){
 for(int j=0;j<fid2d.cols();++j){
 of<<i+1<<" "<<j+1<<" "<<fid2d(i,j).Re()<<" "<<fid2d(i,j).Im()<<std::endl;
 }
 }
 std::cout<<" *Data Saved in '"<<matname
 <<"' (in cols <row> <col> <real> <imag>)"<<std::endl;
 }
}
std::cout<<std::endl;

return(0);
}

```



**--Public Vars**[--- acq1D](#)[--- acq2D](#)**--Constructors**[--- SpinSightStream](#)[--- SpinSightStream](#)**--IO**[--- read](#)**--Other Functions**[--- close](#)[--- is2D](#)[--- is\\_open\(\)](#)[--- open](#)**Examples**[--- converter](#)

```
class SpinSightStream{...
```

---

This is quite a simple class in its function, when given a Directory name, it will determine the necessary parameters to read the binary data contained in the 'data' files into a matrix (2D) or vector (1D). It also gives you access to the elements inside the 'acq' and 'acq\_2' files inside the directory.

The Spin Sight data directory is organized like

expname/

  data --> the binary data

  acq --> the 1D parameters

  acq\_2 --> the 2D auxiliary stuff

  ... --> others that i do not need to know about

Currently it does not write anything because that would require me to know everything about that entire directory structure and the files in it, which i do not....

---

## SpinSight::Public Vars

**Function:**  **Spinsightacq acq1D;**

**Input:**

**Description:**  This is the parameter file parser for the 1D file 'acq'.

**Example Usage:**   
SpinSightStream moo( "mydata/" );  
double dw=0;  
*//get the dwell time*  
moo.acq1D.get( "dw" , dw );  
cout<<dw<<endl;

## SpinSight::Public Vars

**Function:**  **spinSightacq acq2D**

**Input:**

**Description:**  The is the parser object for the 2D parameter file 'acq\_2'.

**Example Usage:**   
SpinSightStream moo( "mydata/ " );  
int td=0;  
//get the number of pts in the  
// second dimension  
moo.acq1D.get( "al2" , td );  
cout<<td<<endl;

## SpinSight::constructor

**Function:**  **SpinSightStream()**

**Input:**  void

**Description:**  Does nothing...to use the class you must use 'open'

**Example Usage:**  `SpinSightStream moo;`  
`moo.open( "myDir/" );`

## SpinSight::constructor

**Function:**       **SpinSightStream(std::string dir);**

**Input:**             dir--> the directory of the data set

**Description:**         This opens up the 'acq' and 'acq\_2' (if present) and the binary data file...

**Example Usage:**     SpinSightStreammoo( "mydir/" );

## SpinSight::IO

**Function:**  `bool read(Vector<complex> &oneD);`  
 `bool read(matrix &twoD);`

**Input:**  oneD--> a complex vector to be filled with 1D data  
twoD--> a complex matrix to be filled with 2D data

**Description:**  This will read the 1D or 2D data from the binary file, converting the endian's as nessesaray into the input containers.

returns 'true' if it worked.

**Example**  `SpinSightStream moo( "mydata/ " );`

**Usage:**  `matrix td;`  
`Vector<complex> od;`  
`if(moo.is2D()) moo.read(td);`  
`else moo.read(od);`

## SpinSight::other

### Function:

□ **void close()**

### Input:

□ void

### Description:

□ Closes all the files and releases all file handles.

### Example Usage:

```
□ SpinSightStream moo("mydata/");
 //this is a boring example
moo.close();
```

## SpinSight::other

### Function:

□ **bool is2D()**

### Input:

□ void

### Description:

□ Returns 'true' if the input directory contains 2D data...

### Example Usage:

```
□ SpinSightStream moo("mydata/");
 if(moo.is2D()){
 cout<<"Yea!! a 2D set..."<<endl;
 }
```

## SpinSight::other

### Function:

□ **bool is\_open()**

### Input:

□ void

### Description:

□ Have we opened up the binary file for reading yet?

### Example Usage:

```
□ SpinSightStream moo("mydata/");
 if(moo.is_open()){
 cout<<"okay i'm open"<<endl;
 }
```

## SpinSight::other

**Function:**  **bool open(std::string dir)**

Input:  dir--> a directory name

Description:  This will open up all the nessesary file for reading: the 'acq', the 'acq\_2' (if present) and the binary data file ('data').

                  returns 'true' if everything worked

Example    SpinSightStream moo;

Usage:     moo.open( "mydata/ " );

Simply converts a SpinSight data file into a Matlab or Text file

---

```
#include "blochlib.h"
//the required 2 namespaces
using namespace BlochLib;
using namespace std;

// Converts binary SpinSight NMR data (BIG ENDIAN)
// to a matlab file for a 1D spectrum

int main(int argc, char **argv)
{
 int q=1;
 std::string fname="", matname;
 std::cout<<std::endl<<"**BlochLib SpinSight (Chemmagnetics) 'directory' --> Matlab converter"<<std::endl;
 query_parameter(argc, argv, q++, " --Enter SpinSight Data Directory: ", fname);
 query_parameter(argc, argv, q++, " --Enter Output File Name: ", matname);
 int txt=0;
 query_parameter(argc, argv, q++, " --Text[0] or Matlab[1] ", txt);

 SpinSightStream loadF(fname);
 double sw,dw;
 int TD, TD2;
 std::string nnuu;

 loadF.acq1D.get("al", TD);
 cout<<" *points in 1D: "<<TD<<endl;

 loadF.acq1D.get("dw", dw);
 sw=1.0/dw;
 cout<<" *Sweep width in 1D: "<<sw<<"Hz"<<endl;

 if(loadF.is2D()){
 loadF.acq2D.get("al2", TD2);
 cout<<" *points in 2D: "<<TD2<<endl;
 }

 std::string vname=matname;
 matstream mm;

 if(txt==1){
 if(matname.find(".mat")>matname.size()){
 matname+=".mat";
 }else{
 vname=matname.substr(0, matname.find(".mat"));
 }
 mm.open(matname, ios::out | ios::binary);
 }
}
```

```

matrix fid2d;
Vector<complex> fid;
if(!loadF.is2D()){
 loadF.read(fid);
 if(txt==1){
 mm.put(vname, fid);
 mm.put("sw1", sw);
 std::cout<< " *Data Saved in '"<<matname
 <<"' under the variable '"<<vname<<"'"<<std::endl;
 std::cout<< " *The Sweep Width information is also stored as variables"
 <<" 'sw1' for 1D info"<<std::endl;
 }else{
 std::ofstream of(matname.c_str());
 for(int i=0;i<fid.size();++i){
 of<<fid[i].Re()<< " <<fid[i].Im()<<std::endl;
 }
 std::cout<< " *Data Saved in '"<<matname
 <<"' (in cols <real> <imag>)"<<std::endl;
 }
}
else{
 loadF.read(fid2d);

 if(txt==1){
 mm.put(vname, fid2d);
 mm.put("sw1", sw);
 std::cout<< " *Data Saved in '"<<matname
 <<"' under the variable '"<<vname<<"'"<<std::endl;
 std::cout<< " *The Sweep Width information is also stored as variables"
 <<" 'sw1' for 1D info"
 <<" 'sw2' is NOT known"<<std::endl;
 }else{
 std::ofstream of(matname.c_str());
 for(int i=0;i<fid.size();++i){
 for(int j=0;j<fid.size();++j){
 of<<i<<" "<<j<<" "<<fid2d(i,j).Re()<< " <<fid2d(i,j).Im()<<std::endl;
 }
 }
 std::cout<< " *Data Saved in '"<<matname
 <<"' (in cols <row> <col> <real> <imag>)"<<std::endl;
 }
}
return(0);
}

```

---

**--Constructors**

--- wavestream

--- wavestream

**--Element Extraction**

--- getBitsPerChannel

--- getBytesPerSecond

---getNumChannels

---getNumSamples

---getNumSeconds

---getSampleRate

**--Assignments**

--- setBitsPerChannel

--- setBytesPerSample

--- setBytesPerSecond

--- setNumChannels

--- setNumSamples

--- setSampleRate

**--IO**

--- close

--- open

--- operator<<

--- operator>>

--- read

--- write

**--Other Functions**

--- ==

--- copyFormat

--- print

--- setupFormat

**Examples**

--- simple reader

class wavestream{ ...

---

This class allows you to read and/or write .wav audio files from/to a data vector. Wave files are either 8 (char) or 16 (short) bits long, however, this class converts any other data type (doubles, ints, floats, etc) into the proper bit stream. You should be careful when converting LARGE floats or ints because a short can only hold ~32,000 as its largest value.

Before you can save a wave file you need to specify the 'sampling rate' (8 kHz, 11.025 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, and 44.1 kHz), the number of channels (1 (mono) or 2 (stereo)), and the bit rate (8 or 16).

Because a wave file needs a special header, the data is not written to the file UNTIL it is either destructed (~wavestream) or closed (wavestream::close()). The header file require the total number of bits, and this can change as you write more and more data. It saves the input data into a valid buffer, then dumps it out to the file...sooooo if your data output is very large, this buffer can get large. If you have the desire to manipulate and write CD length (~700 Mb of data), then i could potentially use a 'temp' file type of creation...if this is desired, please let me know and i will write it up....

If the input of output data is complex it will put (or write) the data as if it was a '2 channel' with the right channel as the real, and the left as the complex values. In reading, the file may not be 2 channels, so only the reals may have any data.

a matrix that is 2xN will be treated as 2 channels as well. A 2xN 'complex' matrix will not work....

## wavestream::constructor

**Function:**  **wavestream()**

**Input:**  void

**Description:**  The empty constructor...does nothing, you need to use 'open' before anything can happen.

**Example Usage:**  wavestream mywave;

## wavestream::constructor

**Function:**  **wavestream(const char \*fname, std::ios::openmode iom);**

**Input:**  fname --> the filename  
 iom--> the open mode

**Description:**  The basic constructor. If iom=std::ios::out, the stream is set up to write, if iom=std::ios::in, the stream is set up to read.

**Example**  wavestream mywave( "moo.wav", std::ios::out );  
**Usage:**

## wavestream::element extraction

**Function:**  **short int getBitsPerChannel() const;**

**Input:**  void

**Description:**  Returns the bits per channel in a file (either 8 or 16)

**Example Usage:** 

```
wavestream mywave;
if(!mywave.open("moo.wav" , std::ios::in)){
 std::cerr<<"cannot open file..."<<std::endl;
}
cout<<mywave.getBitsPerChannel()<<endl;
```

## wavestream::element extraction

**Function:**  **unsigned short getBytesPerSample() const;**

**Input:**  void

**Description:**  Returns the bytes per sample (8 or 16) in the wave file.

**Example Usage:** 

```
wavestream mywave;
if(!mywave.open("moo.wav" , std::ios::in)){
 std::cerr<<"cannot open file..."<<std::endl;
}
cout<<mywave.getBytesPerSample()<<endl;
```

## wavestream::element extraction

|                       |                                                                                                                                                                                            |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b>      | <code>unsigned short getNumChannels() const;</code>                                                                                                                                        |
| <b>Input:</b>         | <code>void</code>                                                                                                                                                                          |
| <b>Description:</b>   | <code>Returns the number of channels in a file.</code>                                                                                                                                     |
| <b>Example Usage:</b> | <pre>wavestream mywave; if( !mywave.open( "moo.wav", std::ios::in)){     std::cerr&lt;&lt;"cannot open file..."&lt;&lt;std::endl; } cout&lt;&lt;mywave.getNumChannels()&lt;&lt;endl;</pre> |

## wavestream::element extraction

**Function:** `unsigned long getNumSamples() const;`

**Input:** `void`

**Description:** `Returns the number of samples in the file (basically the number of unique data words).`

**Example Usage:**

```
wavestream mywave;
if(!mywave.open("moo.wav", std::ios::in)){
 std::cerr<<"cannot open file..."<<std::endl;
}
cout<<mywave.getNumSamples()<<endl;
```

## wavestream::element extraction

**Function:**  **float getNumSeconds() const;**

**Input:**  void

**Description:**  Returns the length (in seconds) of the data file

**Example Usage:** 

```
wavestream mywave;
if(!mywave.open("moo.wav" , std::ios::in)){
 std::cerr<<"cannot open file..."<<std::endl;
}
cout<<mywave.getNumSeconds()<<endl;
```

## wavestream::element extraction

**Function:**  **unsigned long getSampleRate() const;**

**Input:**  void

**Description:**  Returns the sample rate of the wave file (usually 8 kHz, 11.025 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, and 44.1 kHz)

**Example**  wavestream mywave;

**Usage:** if( !mywave.open( "moo.wav" , std::ios::in) ){  
 std::cerr<<"cannot open file..."<<std::endl;  
}  
cout<<mywave.getSampleRate()<<endl;

## wavestream::assignments

**Function:** `void setBitsPerChannel(unsigned short bits);`

**Input:** `bits` --> the number of bits (8 or 16) in a channel

**Description:** `Sets the number of bits per channel (8 or 16)`

**Example Usage:**

```
wavestream mywave;
if(!mywave.open("moo.wav", std::ios::out)){
 std::cerr<<"cannot open file..."<<std::endl;
}
mywave.setBitsPerChannel(16);
```

## wavestream::assignments

**Function:** □ **void setBytesPerSample(unsigned short bytes);**

Input: □ bytes--> the bytes per channel (1 or 2) (8 bit, 16 bit)

Description: □ Sets the number of bytes per sample..1 or 2

Example Usage: □ wavestream mywave;  
if( !mywave.open( "moo.wav", std::ios::out)){  
std::cerr<<"cannot open file..."<<std::endl;  
}  
//a 16 bit output  
mywave.setBytesPerSample(2);

## wavestream::assignments

**Function:** `void setBytesPerSecond(unsigned long bytes);`

**Input:** `bytes` --> a rate of bytes per second

**Description:** `Sets the number of bytes per second (for output)...this should be  
(SampleRate*bytesPerSample*numberChannels)`

**Example** `wavestream mywave;`

**Usage:** `if(!mywave.open("moo.wav", std::ios::out)){  
 std::cerr<<"cannot open file..."<<std::endl;  
}  
//valid for sampleRate=8000 and  
// bytesPerSample=2  
// numberChannels=1  
mywave.setBytesPerSecond(16000);`

## wavestream::assignments

|                  |                                                                                                                                                                                                                                           |
|------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b> | <code>void setNumChannels(unsigned short num);</code>                                                                                                                                                                                     |
| Input:           | <ul style="list-style-type: none"><li>□ num --&gt; the number of channels (1 or 2)</li></ul>                                                                                                                                              |
| Description:     | <ul style="list-style-type: none"><li>□ Sets the number of Channels in the wave file (used in writing)</li></ul>                                                                                                                          |
| Example Usage:   | <ul style="list-style-type: none"><li>□ <pre>wavestream mywave;     if( !mywave.open( "moo.wav" , std::ios::out ) ){         std::cerr&lt;&lt;"cannot open file..."&lt;&lt;std::endl;     }     mywave.setNumChannels(1);</pre></li></ul> |

## wavestream::assignments

**Function:** `void setNumSamples(unsigned long num);`

**Input:** `num` --> the number of samples

**Description:** `Sets the number of samples.` You need not use this function to set this value, as, as you start writing it keeps tally of how many items you have put in the file.

**Example** `wavestream mywave;`

**Usage:** `if (!mywave.open("moo.wav", std::ios::out)){  
 std::cerr<<"cannot open file..."<<std::endl;  
}  
mywave.setNumSamples(16000);`

## wavestream::assignments

**Function:**  **void setSampleRate(unsigned long rate);**

**Input:**  rate --> a valid sampling rate for a wave file.

**Description:**  Sets the sampleing rate of the output file.  
(8 kHz, 11.025 kHz, 16 kHz, 22.05 kHz, 24 kHz, 32 kHz, and 44.1 kHz)

**Example Usage:** 

```
wavestream mywave;
if(!mywave.open("moo.wav" , std::ios::out)){
 std::cerr<<"cannot open file..."<<std::endl;
}
mywave.setSampleRate(8000);
```

## wavestream::IO

**Function:**  **bool close();**

**Input:**  void

**Description:**  Closes the file handle. If the iomode was std::ios::out, then this writes the Header and the data into a proper wave file. Returns true if all is good.

**Example**  wavestream mywave;

**Usage:**  if( !mywave.open( "moo.wav" , std::ios::out ) ){  
 std::cerr<<"cannot open file..."<<std::endl;  
}  
Vector<double> moo(1500, 3);  
mywave<<moo;  
mywave.close();

## wavestream::IO

**Function:** `bool open(const char* name, std::ios::openmode);`

**Input:** `fname` --> the filename  
`iom`--> the open mode

**Description:** `Open` the stream. It does the same thing as the constructor. If `iom=std::ios::out`, the stream is set up to write, if `iom=std::ios::in`, the stream is set up to read. If all goes well, it returns 'true' else false.

**Example** `wavestream mywave;`

**Usage:** `if(!mywave.open("moo.wav", std::ios::out)){  
 std::cerr<<"cannot open file..."<<std::endl;  
}`

## wavestream::IO

**Function:** `□ wavestream &operator<<(wavestream &oo, T out)`

**Input:** `□ oo --> a wave stream`  
`□ T --> a data element or object`

**Description:** `□ The same thing as 'write' except in operator form.`

**Example Usage:** `□ wavestream mywave;`  
`if( !mywave.open( "moo.wav", std::ios::out )){`  
`std::cerr<<"cannot open file..."<<std::endl;`  
`}`  
  
`//use the default`  
`mywave.setupFormat();`  
  
`Vector<double> myDat;`  
`mywave<<myDat<<1.3<<1.2;`

## wavestream::IO

**Function:**  `wavestream &operator<<(wavestream &oo, T &in)`

**Input:**

- oo --> a wave stream
- in --> a data element or object

**Description:**  The same thing as 'read' except in operator form.

**Example Usage:**

```
 wavestream mywave;
 if(!mywave.open("moo.wav", std::ios::in)){
 std::cerr<<"cannot open file..."<<std::endl;
 }

 Vector<double> myDat;
 mywave>>myDat;
```

## wavestream::IO

**Function:**

- `bool read(T &in);`
- `bool read(T *in, int length);`
- `bool read(Vector<T> &in);`
- `bool read(_matrix<T, FullMatrix> &in);`

**Input:**

- `in` --> a data element or object

**Description:**

- Reads a data element..

if the bits per channel=8, then T can only be  
unsigned char, char, float, double, complex<float>, and complex<double>

if bits per channel=16 then T can only be  
short, int, float, double, complex<float>, and complex<double>

when reading complex values it will read single data into the REAL part, and if a 2 channel is present it will read it into the imaginary part

when reading in pointer you need to allocate the memory.

it will attempt to determine the proper size of the object based on the input type 'T' and the number of channels.

if everything functioned properly it will return true

**Example**

- `wavestream mywave;`

**Usage:**

- `if( !mywave.open( "moo.wav" , std::ios::in )){`
- `std::cerr<<"cannot open file..."<<std::endl;`
- `}`

```
Vector<double> myDat;
if(!mywave.read(myDat)){
 cerr<<"error reading file..."<<endl;
}
```

## wavestream::IO

### Function:

```
■ bool write(T &in);
 bool write(T *in, int length);
 bool write(Vector<T> &in);
 bool write(_matrix<T, FullMatrix> &in);
```

### Input:

■ in --> a data element or object

### Description:

■ Reads a data element..

if the bits per channel=8, then T can only be  
unsigned char, char, float, double, complex<float>, and complex<double>

if bits per channel=16 then T can only be  
short, int, float, double, complex<float>, and complex<double>

when writing complex values it will change the number of channels to '2'.

if everything functioned properly it will return true.

if the matrix is 2xN it will change the number of channels to '2'

### Example Usage:

```
■ wavestream mywave;
 if(!mywave.open("moo.wav", std::ios::out)){
 std::cerr<<"cannot open file..."<<std::endl;
 }

 //use the default
 mywave.setupFormat();

 Vector<double> myDat;
 if(!mywave.write(myDat)){
 cerr<<"error writing data..."<<endl;
 }
```

## wavestream::other

**Function:** `bool operator == (const wavestream &lhs)`

**Input:** `lhs` --> another wavestream.

**Description:** `Compares 2 wave files and sees if the formatting matches on both. IT DOES NOT COMPARE the DATA!!!!`

**Example** `wavestream mywave;`

`wavestream mywave2;`

`//this is true;`

`mywave==mywave2;`

`mywave.setNumChannels( 2 );`

`//this is false`

`mywave==mywave2;`

## wavestream::other

**Function:**  **void copyFormat(const wavestream& other);**

**Input:**  other--> another wavestream

**Description:**  Copies the format in one wavestream into another one.

**Example Usage:** 

```
wavestream mywave;
if(!mywave.open("moo.wav" , std::ios::out)){
 std::cerr<<"cannot open file..."<<std::endl;
}

wavestream myw2;
myw2.copyFormat(mywave);
```

## wavestream::other

**Function:** `void print(ostream &out);  
ostream &opeartor<<(ostream &out, wavestream &oo);`

**Input:** `out-->` a ostream to print out on

**Description:** `Prints out all the info we have on the wave file (samples, bits, etc)`

like

Format: 1 (PCM)

Channels: 1

Sample rate: 11025

Bytes per second: 22050

Bytes per sample: 2

Bits per channel: 16

Bytes: 11836

Samples: 5918

Seconds: 0.53678

File pointer: good

**Example Usage:** `wavestream mywave;`

```
if(!mywave.open("moo.wav", std::ios::out)){
 std::cerr<<"cannot open file..."<<std::endl;
}
```

```
//use the default
mywave.setupFormat();

cout<<mywave<<endl;
```

## wavestream::other

**Function:**  **void setupFormat(int sampleRate = 44100, short bitsPerChannel = 16, short channels = 1);**

**Input:**  sampleRate--> the sampling rate (default 44100)  
bitsPerChannel --> the bit per channel (8 or 16)  
channels--> the number of channels (1 or 2)

**Description:**  Sets up the basic format parameters given a sampling Rate (default=44100), the bits perchannel (default=1) and the number of channels (defaul=1). To write a valid wave file.

**Example**  wavestream mywave;

**Usage:** mywave.setupFormat( 8000 );

## A simple wave reader and writer

---

Nothing special...just a nice template to start more complex things from

---

```
#include "blochlib.h"

using namespace BlochLib;
using namespace std;

int main(int argc, const char* argv[])
{
 if (argc <3)
 cout << "reads and writes a wave file...{inwave} {outwave}" << endl;
 else {

 wavestream myW(argv[1], ios::in);

 //dump some info about the input file

 cout<<myW<<endl;
 Vector<short> dat(myW.getNumSamples());

 //reads the data...although it does not
 // do much with it...
 myW>>dat;

 //make a noise 'A' 440 Hz ring
 // 2 seconds long...
 wavestream outW(argv[2], ios::out);

 //out little noise function
 Random<UniformRandom<double> > myR(-0.01,0.01);

 //11025 Hz, 16 bits, and 1 channel
 outW.setupFormat(11025, 16,1);
 //the number of poitns we wish to write
 //(2 seconds worth)
 int len=2*outW.getSampleRate();
 //the frquency
 double freq=440, alpha=0;

 //dump away
 for(int i=0;i<len;++i){
 outW<<(sin(alpha) / 2)+myR();
 alpha+=PI2*freq/outW.getSampleRate();
 }
 //close the final writing
```

```
 outW.close();
}
return 0;
}
```

---

**--Constructors**

--- coord  
--- coord  
--- coord  
--- coord  
--- coord

```
template<class Type=double, int N=3>
class coord{ ...
```

---

**• Accepted Template Params**

- Type --> another class...typically a number type...but can be anything (default=double)
  - N --> the number of dimensions...typically 3 (x,y,z)...Optimizations have been made for N=2, 3, 4 (default=3)
  - Notes:: This is simply a fixed size vector..most mathematical operations are ELEMENT BY ELEMENT except for a few like 'norm,' 'dot,' 'max,' and 'min.' It adds some nice coord transforms also.
  - All the functions valid to 'Vector' are also valid here...the functions created here are optimized for the small size of the coord.
- 

**--Element Extraction**

--- operator()  
--- operator(char)  
--- operator[]  
--- x(),y(),z()

**--Assignments**

--- operator(1,2,...)  
--- operator=  
--- operator=  
--- x(t),y(t),z(t)

**--Math**

--- +, -, /, \*=  
--- +=, -=, /=, \*=  
--- ==, !=, >, >=, <, <=  
--- asin, acos, atan  
--- asinh, acosh, atanh  
--- cart2cyl  
--- cart2sph  
--- ciel, floor, abs  
--- cyl2cart  
--- exp, log  
--- min, max  
--- min, max  
--- norm, dot  
--- sin, cos, tan  
--- sinh, cosh, tanh  
--- sph2cart  
--- sqrt, sqr  
--- ToCyl  
--- ToSphere

**--Other Functions**

--- rotate  
--- Rotate2D  
--- Rotate3D  
--- Vect

## Coordinates::constructor

### Function:

□ `coord(const coord<T1, N> &in);`

### Input:

□ `in-->` another coord

### Description:

□ copy constructor

### Example Usage:

□ `coord<> moo;`  
`coord<> loo(moo);`

## Coordinates::constructor

**Function:** `coord(const Vector<T1> &in);`

Input:  in--> a vector OF LENGTH N

Description:  create a coord from a vector

Example Usage:

- `Vector moo(3,0);`
- `coord<> loo(moo);`

## Coordinates::constructor

### Function:

- `coord<T, 2>(T in1, T in2);`
- `coord<T, 3>(T in1, T in2, T in3);`
- `coord<T, 4>(T in1, T in2, T in3, T in4);`

### Input:

- `in1..in4-->` a 'T' object to fill the coord with in the appropriate position

### Description:

- create initialize and initialize a coord with input values

### Example Usage:

- `coord<> moo(3,4,5); //moo=[3,4,5]`

## Coordinates::constructor

|                  |                                                                             |
|------------------|-----------------------------------------------------------------------------|
| <b>Function:</b> | <code>coord&lt;T, N&gt;(T in)</code>                                        |
| Input:           | <code>in--&gt; a 'T' object</code>                                          |
| Description:     | <code>creates and fills the entire coord up to N with the value 'in'</code> |
| Example Usage:   | <code>coord&lt;&gt; moo(5); //moo=[5,5,5]</code>                            |

## Coordinates::constructor

**Function:** `coord<T, N>(T *data)`

**Input:** `data-->` a pointer to a T object list

**Description:** `creates and fills a coord from a data pointer. It COPIES the memory (not reference).`

**Example Usage:** `double data[ ]={ 2, 3, 4 };`  
`coord<> moo(data); //moo=[ 2, 3, 4 ]`

## Coordinates::element extraction

### Function:

□ **Type operator(int);**

### Input:

□ an int from 0--N

### Description:

□ Extracts an element from the coord

### Example Usage:

□ `double firstelement=oldC(0);`

## Coordinates::element extraction

### Function:

□ **Type operator(char)**

### Input:

□ a character 'x','y','z','t'

### Description:

□ Extracts an element given 'x'==0, 'y'==1, 'z'==2, 't'==3

### Example Usage:

□ `double myX=oldC('x');`

## Coordinates::element extraction

### Function:

**Type operator[int]**

### Input:

none

### Description:

Extracts an element from the coord

### Example Usage:

double secondele=oldC[1];

## Coordinates::element extraction

### Function:

- Type &x()** or **Type &r()**
- Type &y()** or **Type &theta()**
- Type &z()** or **Type &phi()**
- Type &t()**

### Input:

- none

### Description:

- x()->returns the 1st element of the coord
- y()->returns the 2nd element of the coord
- z()->returns the 3rd element of the coord
- t()->returns the 4th element of the coord

### Example Usage:

- `double xele=oldC.x(); //extraction`
- `oldC.x()=8; //assignment`

## Coordinates::assignments

### Function:

- `void operator(T x, T y)`
- `void operator(T x, T y, T z)`
- `void operator(T x, T y, T z, T t)`

### Input:

- `x-->a number of type T to go in the first element`
- `y --> a number of type T to go in the second element`

### Description:

- `(T x, T y)->Sets the first 2 elements of a coord...will attempt a type conversion`
- `(T x, T y, T z) ->Sets the first 3 elements of a coord...will attempt a type conversion`
- `(T x, T y, T z, T t)->Sets the first 4 elements of a coord...will attempt a type conversion`

Example Usage: □ `oldC(2,3); //the first 2 elements of oldC are now 2 and 3`

## Coordinates::assignments

- Function:**
  - `coord<Type, N> operator=(const coord<T, N> &rhs)`
- Input:**
  - rhs-->a coord object OF THE SAME LENGTH as left-hand-side(can be of different type)
- Description:**
  - Assignment operator...there is NO 'N' conversions assigment...does do Type conversions
- Example Usage:**
  - `newC=oldC;`

## Coordinates::assignments

**Function:** □ `coord<Type, N> operator=(const Vector<T> &rhs)`

**Input:** □ rhs-->a Vector object OF THE SAME LENGTH as N (can be of different type)

**Description:** □ Assignment operator...does do Type conversions

**Example Usage:** □ `Vector<int> tmpVec(3, 2);  
oldC=tmpVec;`

## Coordinates::assignments

### Function:

- **void x(T in)** or **void r(T in)**
- void y(T in)** or **void theta(T in)**
- void z(T in)** or **void phi(T in)**
- void t(T in)**

### Input:

- in--> A number of type T(int, double, etc...or another object)

### Description:

- x(T)->sets the 1st element of the coord with inx (doing type conversions to type 'Type')
- y(T)->sets the 2nd element of the coord with inx (doing type conversions to type 'Type')
- z(T)->sets the 3rd element of the coord with inx (doing type conversions to type 'Type')
- t(T)->sets the 4th element of the coord with inx (doing type conversions to type 'Type')

### Example Usage:

- oldC.y( 45 );

## Coordinates::Math

**Function:**

- `coord<Type, N> operator+=(coord<Type, N> &rhs)`
- `coord<Type, N> operator+=(Vector<T> &rhs)`
- `coord<Type, N> operator+=(T &rhs)`
  
- `coord<Type, N> operator-=(coord<Type, N> &rhs)`
- `coord<Type, N> operator-=(Vector<T> &rhs)`
- `coord<Type, N> operator-=(T &rhs)`
  
- `coord<Type, N> operator/=(coord<Type, N> &rhs)`
- `coord<Type, N> operator/=(Vector<T> &rhs)`
- `coord<Type, N> operator/=(T &rhs)`
  
- `coord<Type, N> operator*=(coord<Type, N> &rhs)`
- `coord<Type, N> operator*=(Vector<T> &rhs)`
- `coord<Type, N> operator*=(T &rhs)`

**Input:**

- `rhs->a coord, Vector or scalar (number)`

**Description:**

- self addition(+=), subtraction(-=), division(/=), or multiplication(\*=) operator with another coord, Vector, or constant.

-For a coord with N=3...

`a+=b--> a[0]+=b[0]; a[1]+=b[1]; a[2]+=b[2];`

-For a Vector of size N=3...

`a+=b--> a[0]+=b[0]; a[1]+=b[1]; a[2]+=b[2];`

-For a Vector of size N=3...

`a+=4--> a[0]+=4; a[1]+=4; a[2]+=4;`

**Example**

- `oldC+=newC*2+4;`
- `oldC+=tmVec*2+4;`
- `oldC+=4;`

**Usage:**

## Coordinates::Math

**Function:**

- `coord<Type,N> operator+(const coord<T1,N> &lhs, const coord<T2,N> &rhs)`
- `coord<Type,N> operator+(const coord<T1, N> &rhs, const Vector<T2> &lhs)`
- `coord<Type,N> operator+(const coord<T1, N> &rhs, const Vector<T2> &lhs)`
- `coord<Type,N> operator+(const coord<T1,N> &lhs, const T2 rhs)`  
*(similar for -, /, and \*)*

**Input:**

- lhs--> A coord of length N (the left hand side)

- rhs-->A coord, Vector, or Number of type 'T' (the right hand side)

**Description:**

- Adds, Subtracts, Divides, or Multiplies the Number 'rhs' ELEMENT BY ELEMENT to the 'lhs'  
Will convert to the 'highest' type between T1 and T2 (i.e. int+double-->double)

**Example**

```
□ coord<> c1(4);
 coord<> c2;
 Vector<int> v2(3, 2);
 c2=c1+v2; // the result is now a double coord with elements 6
 c2=v2+c1; // also valid with elements as 6
```

**Usage:**

## Coordinates::Math

**Function:**

- `bool operator==(coord<Type, N> &rhs)`
- `bool operator!=(coord<Type, N> &rhs)`
- `bool operator>=(coord<Type, N> &rhs)`
- `...similar from other operators`

**Input:**

- `coord<Type, N>`

**Description:**

- the comparison operators....

performs the initial comparison element by element and if any of the elements if false it will return false, if non of the elements are false it will return true

if `a=(1,2,2)` and `b=(3,1,3)`..then `b>a` will be FALSE

if `a=(1,2,2)` and `b=(3,3,3)`..then `b>a` will be TRUE

**Example**

- `bool test= oldC!=newC;`

**Usage:**

- `bool test= oldC>newC;`

## Coordinates::Math

- Function:**
- `coord<Type, N> asin(coord<Type, N> &rhs)`
  - `coord<Type, N> acos(coord<Type, N> &rhs)`
  - `coord<Type, N> atan(coord<Type, N> &rhs)`
- Input:**
- `coord<Type, N>`
- Description:**
- performs an element-by-element evalutation of the inverse trig functions asin, acos, atan
- Example Usage:**
- `newC=acos(oldC*Pi/180)`

## Coordinates::Math

**Function:**  `coord<Type, N> asinh(coord<Type, N> &rhs)`  
 `coord<Type, N> acosh(coord<Type, N> &rhs)`  
 `coord<Type, N> atanh(coord<Type, N> &rhs)`

Input:  `coord<Type, N>`

Description:  performs an element-by-element evalutation of the inverse hyperbolic trig functions asinh, acosh, atanh

Example  `newC=atanh(oldC*Pi/180)`

Usage:

## Coordinates::Math

**Function:**  `coord<Type,N> cart2cyl()`

**Input:**  none

**Description:**  valid for N=3 only  
returns the coordinate in cylindrical coords from assuming an existing cartesian coord

**Example Usage:**  `newC=oldC.cart2cyl();`

## Coordinates::Math

**Function:**  `coord<Type,N> cart2sph()`

Input:  none

Description:  valid for N=3 only  
returns the coordinate in cartesian coords from assuming an existing spherical coord

Example Usage:  `newC=oldC.cart2sph();`

## Coordinates::Math

**Function:**  `coord<Type, N> ceil(coord<Type, N> &rhs)`  
 `coord<Type, N> floor(coord<Type, N> &rhs)`  
 `coord<Type, N> abs(coord<Type, N> &rhs)`

Input:  `coord<Type, N>`

Description:  element-by-element of 'ceil' (if  $n=4.9$ ,  $\text{ceil}(n)=5$ ), 'floor' (if  $n=4.6$   $\text{floor}(n)=4$ ), and 'abs' (if  $n=-9$ ,  $\text{abs}(n)=9$ )

Example  `newC=floor(oldC)`  
Usage:

## Coordinates::Math

**Function:**  `coord<Type,N> cyl2cart()`

**Input:**  none

**Description:**  valid for N=3 only  
returns the coordinate in cartesian coords from assuming an existing cylindrical coord

**Example Usage:**  `newC=oldC.cyl2cart();`

## Coordinates::Math

### Function:

- `coord<Type, N> exp(coord<Type, N> &rhs)`
- `coord<Type, N> log(coord<Type, N> &rhs)`

### Input:

- `coord<Type, N>`

### Description:

- element-by-element evaluation of exponential and natural logarithm

### Example Usage:

- `newC=exp(oldC)`

## Coordinates::Math

### Function:

- `Type min(coord<Type, N> &rhs)`
- `Type max(coord<Type, N> &rhs)`

### Input:

- `coord<Type, N>`

### Description:

- finds the max (or min) value inside the coord

### Example Usage:

- `double mymin=min(oldC)`

## Coordinates::Math

**Function:**  **Type** `min(coord<T1, N> &rhs, T2 num)`  
 **Type** `max(coord<T1, N> &rhs, T2 num)`

**Input:**  rhs-->A coord of the Type T1  
 num--> a number of type T2

**Description:**  eturns the min (or max) entry in the list in rhs and then the smallest (or largest) between 'num' and that number

**Example**  `coord<> c1(6);`

**Usage:**  `double mymin=min(c1, 0.2); //the result here is 0.2`  
`double mymin=min(0.2, c1/100); //this is also valid with the result of 0.06`

## Coordinates::Math

- Function:**
- **Type** `dot(coord<Type, N> &c1 coord<Type2, N> &c2)`
  - **Type** `norm(coord<Type, N> &rhs)`
- Input:**
- `rhs-->`Another coord of the SAME Type
- Description:**
- norm returns (if N=3)  $(x^*x + y^*y + z^*z)^{(1/2)}$
  - dot returns  $(x_1*x_2 + y_1*y_2 + z_1*z_2)^{(1/2)}$
- Example Usage:**
- `double mynorm=norm(oldC);`
  - `double dd=dot(oldC, newC);`

## Coordinates::Math

- Function:**
- `coord<Type, N> sin(coord<Type, N> &rhs)`
  - `coord<Type, N> cos(coord<Type, N> &rhs)`
  - `coord<Type, N> tan(coord<Type, N> &rhs)`
- Input:**
- `coord<Type, N>`
- Description:**
- performs an element-by-element evalutation of the trig functions sin, cos, tan
- Example Usage:**
- `newC=cos(oldC*Pi/180);`

## Coordinates::Math

### Function:

- `coord<Type, N> sinh(coord<Type, N> &rhs)`
- `coord<Type, N> cosh(coord<Type, N> &rhs)`
- `coord<Type, N> tanh(coord<Type, N> &rhs)`

### Input:

- `coord<Type, N>`

### Description:

- performs an element-by-element evalutation of the hyperbolic trig functions sinh, cosh, tanh

### Example Usage:

- `newC=tanh(oldC*pi/180)`

## Coordinates::Math

**Function:**  `coord<Type,N> sph2cart()`

**Input:**  none

**Description:**  valid for N=3 only  
returns the coordinate in cartesian coords from assuming an existing spherical coord

**Example Usage:**  `newC=oldC.sph2cart();`

## Coordinates::Math

|                       |                                                                                                                                                                                                        |
|-----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b>      | <ul style="list-style-type: none"><li>□ <code>coord&lt;Type, N&gt; sqrt(coord&lt;Type, N&gt; &amp;rhs)</code></li><li>□ <code>coord&lt;Type, N&gt; sqr(coord&lt;Type, N&gt; &amp;rhs)</code></li></ul> |
| <b>Input:</b>         | <ul style="list-style-type: none"><li>□ <code>coord&lt;Type, N&gt;</code></li></ul>                                                                                                                    |
| <b>Description:</b>   | <ul style="list-style-type: none"><li>□ element-by-element square and square root</li></ul>                                                                                                            |
| <b>Example Usage:</b> | <ul style="list-style-type: none"><li>□ <code>newC=sqr(oldC)</code></li></ul>                                                                                                                          |

## Coordinates::Math

**Function:**  **void ToCyl()**

Input:  none

Description:  valid for N=3 only

alters the coordinate (any type) into cartesian coords from an assumed cartesian coord  
this is a destructive conversion....it is faster then the above converstions, but you loose the other information

$r=(\sqrt{x*x+y*y});$

$\theta=(fmod(\tan(y/x), PI2));$

Example  oldC.ToCart();

Usage:

## Coordinates::Math

**Function:**  **void ToSphere()**

Input:  none

Description:  valid for N=3 only

alters the coordinate (any type) into spherical coords from assuming a cartesian...this is a destructive conversion....it is faster then the above converstions, but you loose the other information

```
r=(sqrt(x*x+y*y+z*z));
phi(asin(z/r()));
theta(acos(x/(r()*cos(phi()))));
```

Example  oldC.ToSphere();

Usage:

## Coordinates::other

**Function:** `void rotate(double theta, const coord<> &axis)`

**Input:**

- theta --> the rotation angle in RADIANS
- axis --> the rotaion axis about which the coord will be rotated (IT MUST BE NORMALIZED)

**Description:**

- Rotates a 3 VECTOR by an angle theta (in radians) around the axis (axis).

**Example Usage:** `coord<> moo(3,2,4);`

```
moo.rotate(34*DEG2RAD, coord<>(0,0,1);
```

## Coordinates::other

**Function:**  **void Rotate2D(double theta, double phi=0)**

**Input:**  theta-->an angle in radians (rotation about 'z-axis')

phi --> an angle in radians (x-yplane rotation) (default=0)

**Description:**  Performs a 2D rotation on the current coord through angles (theta, phi) (in radians)... will transform the first 2 elements leaves all other untouched

**Example**  oldC.Rotate2D(45\*Pi/180, 70\*Pi/180);

**Usage:**  oldC.Rotate2D(45\*Pi/180); //uses phi=0 default

## Coordinates::other

**Function:** `void Rotate3D(double theta, double phi=0, double psi=0)`

**Input:** `theta`-->an angle in radians (angle from z-axis rotation)

`phi`--> an angle in radians (x-yplane rotation) (default=0)

`psi`--> an angle in radians (z phase rotation) (default=0)

**Description:** `Performs a 3D rotation on the current coord through angles (theta, phi, psi) (in radians)... will transform the first 3 elements leaves all other untouched`

**Example** `oldC.Rotate3D(45*Pi/180, 70*Pi/180, 90*Pi/180);`

**Usage:** `oldC.Rotate3D(45*Pi/180, 70*Pi/180); //uses default for psi`  
`oldC.Rotate3D(45*Pi/180); //uses default for phi and psi`

## Coordinates::other

### Function:

□ `Vector<Type> Vect()`

### Input:

□ none

### Description:

□ returns the Vector equivalent of the coord...will be of length N...

### Example Usage:

□ `Vector<double> newV=oldC.Vect();`

|                      |                                                                                                                                                                              |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| --Global Constants   | class Range{....                                                                                                                                                             |
| --- Start, End       |                                                                                                                                                                              |
| --Constructors       | template<class T>                                                                                                                                                            |
| --- Range,Spread     | class Spread{...                                                                                                                                                             |
| --- Range,Spread     |                                                                                                                                                                              |
| --- Range,Spread     |                                                                                                                                                                              |
| --- Range,Spread     |                                                                                                                                                                              |
| --Element Extraction | Both Range and Spread are essentially the same thing... however, Range uses the 'INT' class type.                                                                            |
| --- operator(int)    | Range is meant for accessing elements in arraies/vector/etc, with either non-uniform steps sizes and/or sub elements                                                         |
| --- operator[int]    | The Vector and Matrix classes have the operator(Range) defined for you to access the sub elements                                                                            |
| --Assignments        |                                                                                                                                                                              |
| --- operator=        | Spread is meant to create quick 'lists' of numbers with arbitrary start, stop and step sizes..and NOT for element accessing (its hard to access the '0.1' element in a list) |
| --IO                 |                                                                                                                                                                              |
| --- operator<<       | Exact implementations of 'Range' for each container can be found in its own class description....                                                                            |
| --Other Functions    |                                                                                                                                                                              |
| --- +, -             |                                                                                                                                                                              |
| --- begin, first     |                                                                                                                                                                              |
| --- last,end         |                                                                                                                                                                              |
| --- stride           |                                                                                                                                                                              |

## Range/Spread::global constant

### Function:

- Range::Start**
- Range::End**

### Input:

- na

### Description:

- These are enum values that allow you to span the entire list of elements.  
ia a=(1,2,3,4);  
a(Range(Range::Start, 2))=(1,2,3);  
a(Range(Range::Start,Range::End))=(1,2,3,4);  
a(Range(2, Range::End)))=(3,4);

### Example Usage:

- b=a(Range(Range::Start, 2)); //=(1,2,3);  
b=a(Range(Range::Start,Range::End)); //=(1,2,3,4);  
b=a(Range(2, Range::End))); //=(3,4);

## Range/Spread::constructor

- Function:**  **Range()**,  
 **Spread()**
- Input:**  None
- Description:**  --empty constructor--> Sets up the default Range  
Range(Range::Start,Range::End,1)...basically a range that holds every elements  
--for Spread(0,1)-->Start at 0 goes to 1 with a '1' stepsize
- Example Usage:**  Range moo; //this does nothing until it is used  
 Spread<int> loo; //creates the list (0..1)

## Range/Spread::constructor

- Function:**
- `Range(int start, int end)`
  - `Spread<T>(T start, T end)`
- Input:**
- start--> the starting value in the list
  - end--> the ending value in the list
- Description:**
- creates the list going from start to end with a '1' step size
- Example Usage:**
- `Range moo(0, 30); //a list from (0..1..2....29..30)`
  - `Spread<double> loo(-6, 0); //a list from (-6...-5....-1..0)`

## Range/Spread::constructor

**Function:**  `Range(int start, int end, int step),`  
`Spread<T>(T start, T end, T step)`

**Input:**  start--> the starting value in the list  
end--> the ending value in the list  
step--> the step size

**Description:**  creates the list going from start to end with a 'step' step size

**Example**  `Range moo(0, 30, 3); //a list from (0..3..6...27..30)`

**Usage:**  `Spread<double> loo(-6.0, 0, 0.5); //a list from (-6..-5.5....-0.5..0)`

## Range/Spread::constructor

### Function:

- `Range(const Range &cp),`
- `Spread<T>(const Spread<T> &cp),`
- `Spread<T>(const Range &cp)`

### Input:

- `cp-->` an object to copy...

### Description:

- creates the list which copies the one from 'cp'
- Spread can also take in a Range

### Example Usage:

- `Range moo(0, 30); //a list from (0..1..2....29..30)`
- `Spread<double> loo(moo); //a list from (0..1..2....29..30)`

## Range/Spread::element extraction

### Function:

- `int operator(int i) (for Range),  
T operator(int i) (for Spread)`

### Input:

- `i->element in the Range/Spread List`

### Description:

- Returns the i'th element in the list as calculated from (first + i \* stride);

### Example Usage:

- ```
Range moo(2,10,2);
int ele=moo(3); //returns '6'
Spread<double> loo(-0.4, 0.9, 0.1);
double ele2=loo(7); //returns '0.2'
```

Range/Spread::element extraction

Function:

- `int operator[int i] (for Range),
T operator[int i](for Spread)`

Input:

- `i` --> element in the Range/Spread List

Description:

- Returns the `i`'th element in the list as calculated from (`first + i * stride`);

Example Usage:

- `Range moo(2,10,2);
int ele=moo[3]; //returns '6'
Spread<double> loo(-0.4, 0.9, 0.1);
double ele2=loo[7]; //returns '0.2'`

Range/Spread::assignments

Function:

- `Range &operator=(Range &rhs);`
- `Spread<T> &operator=(Spread<T> &rhs);`
- `Spread<T> &operator=(Range &rhs);`

Input:

- rhs--> the item you wish to copy...

Description:

- assigns one range/spread from another...(a hard copy...i.e. dupes the memory)

Example Usage:

- `Range moo(2,10,2);`
- `Spread<double> loo(-0.4, 0.9, 0.1);`
- `Sread<double> koo;`
- `koo=loo;`
- `koo=moo;`

Range/Spread::IO

Function: `ostream &operator<<(Range &r)`

Input: `r-->` the Range/Spread you wish to output

Description:

- ❑ writes the string....
`'Range(<first>, <last>, <stride>)'`

Example Usage: `Range moo(2,10,2);
cout<<moo; // will write the string "Range(2,10,2)"`

Range/Spread::other

Function:

□ Range operator+(int &rhs); (for Range)
Spread<T> operator+(T rhs); (for Spread<T>)

Input:

□ rhs-->a number to increment the Range by

Description: □ returns a new Range/Spread with a new first and last range given by first+rhs..last+rhs

Example Usage: □ Spread<int> moo(2,10,2); //a list from (2..4..6..8..10)
Spread<int> loo=moo+2; //a list from (4..6..8..10.12)
//----
Spread<int> moo(2,10,2); //a list from (2..4..6..8..10)
Spread<int> loo=moo-5; //a list from (-3..-1..1..3..5)

Range/Spread::other

Function:	<ul style="list-style-type: none">□ <code>int begin(); (for Range)</code>□ <code>T begin(); (for Spread<T>)</code>□ <code>int first(); (for Range)</code>□ <code>T first(); (for Spread<T>)</code>
Input:	<ul style="list-style-type: none">□ none
Description:	<ul style="list-style-type: none">□ returns the last element in the range/spread (same as 'first')
Example Usage:	<ul style="list-style-type: none">□ <code>Spread<int> moo(2,10,2);</code>□ <code>Spread<int>::numtype fir=moo.begin(); //will return '2'</code>

Range/Spread::other

Function:	<ul style="list-style-type: none">□ <code>int last(); (for Range)</code>□ <code>T last(); (for Spread<T>)</code>□ <code>int end(); (for Range)</code>□ <code>T end(); (for Spread<T>)</code>
Input:	<ul style="list-style-type: none">□ none
Description:	<ul style="list-style-type: none">□ returns the last element in the range/spread
Example Usage:	<ul style="list-style-type: none">□ <code>Spread<int> moo(2,10,2);</code>□ <code>Spread<int>::numtype fir=moo.last(); //will return '10'</code>

Range/Spread::other

Function:	<code>int stride(); (for Range)</code> <code>T stride(); (for Spread<T>)</code>
Input:	<input type="checkbox"/> none
Description:	<input type="checkbox"/> returns the step size of the range/spread (same as last)
Example Usage:	<input type="checkbox"/> <code>Spread<int> moo(2,10,2);</code> <code>Spread<int>::numtype fir=moo.stride(); //will return '2'</code>

--Constructors	template<class Type> class Vector : public MemChunk<Type>{....
--Vector	Accepted Template Params Type --> another class...typically a number type (int, double, complex, float....)...but can be anything (default=NONE)
--Vector	MemChunk is a class that handles the memory allocation and deallocation for ALL the various containers here
--Vector	Notes:: mathematical operations are ELEMENT BY ELEMENT except for a few like 'norm,' 'dot,' 'max,' and 'min.'
--Element Extraction	Matrix--Vector operations are described in 'Matrix'
--- get	
--- get	
--- operator()	
--- operator(Range)	
--- operator[Range]	
--- operator[]	
--- put	
--- put	
--Assignments	!!NOTE!!
--- operator=	NUMERICAL NOTE::DO NOT confuse this class with the STL 'vector' (the one here has a CAPITAL 'V'). The STL vector class is much better suited as a 'holder' or an object that simply holds a bunch of objects...this class is meant for performing numerical operations on large lists...it does maintain the same resizing/adding/dropping options as the STL version, but they are a bit slower than the STL version
--- operator=	
--- operator=	
--Math	
--- +=, -=, /=, *=	NUMERICAL NOTE:: The default compiling options DOES NOT CHECK ANY BOUNDS...if you go outside the Vectors length you will crash the program...this is for SPEED...if it does not have to check the bounds then it is one less operation it must do. YOU CAN TURN THIS BACK ON by defining 'VBoundCheck 1' in the compile string
--- +, -, /, *	i.e. g++ -DVBoundCheck myprog.cc
--- ==, !=, >, >=, <, <=	
--- asin, acos, atan	
--- asinh, acosh, atanh	
--- ciel, floor, abs	
--- exp, log	
--- fft, ifft	
--- fftshift, ifftshift	
--- min, max	
--- min, max	
--- norm, dot	
--- sin, cos, tan	
--- sinh, cosh, tanh	
--- sqrt, sqr	
--- sum	
--Other Functions	
--- apply	
--- empty	
--- fill	
--- pop	
--- push_back	
--- resize	
--- resizeAndPreserve	
--- rotateRight	
--- size, length	

Vector::constructor

Function:

Vector()

Input:

None

Description:

empty constructor, sets length=0

Example Usage:

`Vector<double> moo;`

Vector::constructor

Function: `Vector(const Vector<T> &cp)`

Input: another Vector of number Type 'T' and length 'N1'

Description: copy constructor
performs auto type conversions from 'T' to 'Type' if possible else error at compile time

Example Usage: `Vector<double> moo2(moo);`

Vector::constructor

Function: `□ Vector(int num)`

Input: `□ num--> the Length of the new Vector`

Description: `□ Creates a Vector of length 'num' will all elements set to Type(0)..`

Example Usage: `□ Vector<double> moo2(3);`

Vector::constructor

Function: `□ Vector(Range R)`

Input: `□ R--> A range from [start...end]`

Description: `□ Creates a Vector of length of the range that simply is the entire range in integer form`

Example Usage: `□ Vector<double> moo2(Range(3,4)); //produces [3,4]`
`□ Vector<int> moo(Range(-4, 2, 2)); //produces [-4,-2,0,2]`

Vector::element extraction

Function:

□ **Type get(int R);**

Input:

□ R->an int from 0..size()

Description:

□ Extracts an element from the vector

Example Usage:

□ `double tmm=oldC.get(3); //extraction`

Vector::element extraction

Function: `□ Type get(Range R);`

Input: `□ R->a Range`

Description: `□ Extracts an subvector from the vector`

Example

`□ Vector<double> tmm=oldC.get(Range::Start, 3)); //extraction`

Usage:

Vector::element extraction

Function:

□ **Type operator(int);**

Input:

□ an int from 0--size()

Description:

□ Extracts an element from the coord

Example Usage:

□ `double firstelement=oldC(0);`

Vector::element extraction

Function: `□ Vector<T> &operator(Range R);`

Input: `□ R--> A range`

Description: `□ Extracts a sub vector of a vector
can also be used for assignment`

Example Usage: `□ Vector<double> V1(8, 1); V1[3]=9; // V1=[1,1,1,9,1,1,1,1]
Vector<double> V2=V1[Range(2, 5)]; // V2=[1,9,1,1]
V2(Range(0, 2))=V1(Range(1, 3)); //V2=[1,1,9,1]`

Vector::element extraction

Function: `□ Vector<T> &operator[Range R];`

Input: `□ R--> A range`

Description: `□ Extracts a sub vector of a vector
can also be used for assignment`

Example Usage: `□ Vector<double> V1(8, 1); V1[3]=9; // V1=[1,1,1,9,1,1,1,1]
Vector<double> V2=V1[Range(2, 5)]; // V2=[1,9,1,1]
V2[Range(0,2)]=V1[Range(1,3)]; //V2=[1,1,9,1]`

Vector::element extraction

Function:

Type operator[int]

Input:

none

Description:

Extracts an element from the coord

Example Usage:

double secondele=oldC[1];

Vector::element extraction

Function: `void put(int where, const T num)`

Input:

- where --> an integer
- num-->an number of Type 'T'

Description:

- puts the element 'num' in the position 'where'

Example Usage: `Vector<int> tmpVec(3, 2);
tmpVec.put(1,5); //puts the number '5' in element 1`

Vector::element extraction

Function: `void put(Range R, Vector<T1> rhs)`

Input:

- R -->the range to place the items in 'rhs' into the vector
- rhs-->A vector that is AT LEAST as long as the range, R, (it can be longer)

Description:

- puts the elements begining at '0' in rhs in the posistions of R.begin into the output vector

Example Usage:

- `Vector<int> V1(6, 2); //V1=[2,2,2,2,2,2]`
- `V1.put(Range(3,5),Vector<int>(3,4)); V1=[2,2,2,4,4,4]`

Vector::assignments

Function: `Vector<T> operator=(const Vector<T2> &rhs)`

Input: rhs-->a Vector object OF THE SAME LENGTH as left-hand-side(can be of different type)

Description: Assignment operator...there is NO 'N' conversions assigment...does do Type conversions

Example Usage: `newC=oldC;`

Vector::assignments

- Function:**
 - `Vector<T> operator=(const coord<T,N> &rhs)`
- Input:**
 - `rhs-->a coord<T,N> object`
- Description:**
 - Assignment from a coord operator...does do Type conversions
the resulting vector will be of size()=N
- Example Usage:**
 - `coord<> cor(3, 2);`
 - `Vector<double> tmpVec=cor;`

Vector::assignments

Function: `□ Vector<Type> operator=(T &rhs)`

Input: `□ rhs->a number`

Description: `□ will fill the entire vector the the 'rhs' (will not change the vector's length to 1)`

Example Usage: `□ Vector<> moo(5);
moo=7; // moo=[7,7,7,7,7];`

Vector::Math

Function:

- `Vector<T> operator+=(Vector<T> &rhs)`
- `Vector<T> operator+=(coord<T1, N> &rhs)`
- `Vector<T> operator+=(T &rhs)`

- `Vector<T> operator-=(Vector<T> &rhs)`
- `Vector<T> operator-=(Vector<T> &rhs)`
- `Vector<T> operator-=(T &rhs)`

- `Vector<T> operator/=(Vector<T> &rhs)`
- `Vector<T> operator/=(Vector<T> &rhs)`
- `Vector<T> operator/=(T &rhs)`

- `Vector<T> operator*=(Vector<T> &rhs)`
- `Vector<T> operator*=(Vector<T> &rhs)`
- `Vector<T> operator*=(T &rhs)`

Input:

- rhs->a coord, Vector or scalar (number)

Description:

- self addition(+=), subtraction(-=), division(/=), or multiplication(*=) operator with another coord, Vector, or constant.

-For a coord with size()=3...

`a+=b--> a[0]+=b[0]; a[1]+=b[1]; a[2]+=b[2];`

-For a Vector of size N=3...

`a+=b--> a[0]+=b[0]; a[1]+=b[1]; a[2]+=b[2];`

-For a Vector of size N=3...

`a+=4--> a[0]+=4; a[1]+=4; a[2]+=4;`

Example

- `oldC+=newC*2+4;`
- `oldC+=tmVec*2+4;`
- `oldC+=4;`

Usage:

Vector::Math

Function:

- `Vector<T> operator+(const Vector<T1> &rhs, const Vector<T2> &lhs)`
- `Vector<T> operator+(const Vector<T1> &rhs, const coord<T2,N> &lhs)`
- `Vector<T> operator+(const Vector<T1> &rhs, const T2 rhs)`
*(similar for -, /, and *)*

Input:

- `lhs`--> A Vector of length N (the left hand side)
- `rhs`-->A coord, Vector, or Number of type 'T' (the right hand side)

Description:

- Adds, Subtracts, Divides, or Multiplies the Number 'rhs' ELEMENT BY ELEMENT to the 'lhs'
Will convert to the 'highest' type between T1 and T2 (i.e. int+double-->double)

Example

```
□ Vector<double> c1(4);  
Vector<double> c2;  
Vector<int> v2(3, 2);  
c2=c1+v2; // the result is now a double Vector with elements 6  
c2=v2+c1; // also valid with elements as 6
```

Usage:

Vector::Math

Function:

- `bool operator==(Vector<T> &rhs)`
- `bool operator!=(Vector<T> &rhs)`
- `bool operator>=(Vector<T> &rhs)`
- `...similar from other operators`

Input:

- `Vector<T>`

Description:

- the comparison operators....

performs the initial comparison element by element and if any of the elements if false it will return false, if non of the elements are false it will return true

if `a=(1,2,2)` and `b=(3,1,3)`..then `b>a` will be FALSE

if `a=(1,2,2)` and `b=(3,3,3)`..then `b>a` will be TRUE

Example

- `bool test= oldC!=newC;`

Usage:

- `bool test= oldC>newC;`

Vector::Math

- Function:**
- `Vector<T> asin(Vector<T> &rhs)`
 - `Vector<T> acos(Vector<T> &rhs)`
 - `Vector<T> atan(Vector<T> &rhs)`
- Input:**
- `Vector<T>`
- Description:**
- performs an element-by-element evalutation of the inverse trig functions asin, acos, atan
- Example Usage:**
- `newC=acos(oldC*Pi/180)`

Vector::Math

Function:

- `Vector<T> asinh(Vector<T> &rhs)`
- `Vector<T> acosh(Vector<T> &rhs)`
- `Vector<T> atanh(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- performs an element-by-element evalutation of the inverse hyperbolic trig functions asinh, acosh, atanh

Example

- `newC=atanh(oldC*Pi/180)`

Usage:

Vector::Math

Function:

- `Vector<T> ceil(Vector<T> &rhs)`
- `Vector<T> floor(Vector<T> &rhs)`
- `Vector<T> abs(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- element-by-element of 'ceil' (if $n=4.9$, $\text{ceil}(n)=5$), 'floor' (if $n=4.6$ $\text{floor}(n)=4$), and 'abs' (if $n=-9$, $\text{abs}(n)=9$)

Example

- `newC=floor(oldC)`

Usage:

Vector::Math

Function:

- `Vector<T> exp(Vector<T> &rhs)`
- `Vector<T> log(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- element-by-element evaluation of exponential and natural logarithm

Example Usage:

- `newC=exp(oldC)`

Vector::Math

Function: `Vector<complex> fft(const Vector<T> &in)`
 `Vector<complex> ifft(const Vector<T> &in)`

Input: `in` --> a `Vector<double>` or `Vector<complex>`

Description: Return the complex Fourier transform (fft) or the inverse (ifft) of the in vector. T can be double, or complex. It uses FFTW if it is available, and returns a vector where the order of the fft elements is 0...w...-w

to get the 'spectral' ordering (-w...0...w) use 'fftshift'

Example `Vector<double> myV(5 , 2);`

Usage: `Vector<complex> myFFT=fftshift(fft(myV));`

Vector::Math

Function: `Vector<T> fftshift(const Vector<T> &in)`
`Vector<T> ifftshift(const Vector<T> &in)`

Input: `Vector<T>` --> a vector of any type

Description: Simply 'rotates' the Vector....places the last half of the vector as the first half, to achieve the 'spectral' ordering seen in most spectra. ifftshift returns the vector to the normal (algorithmic) ordering before on should perform an ifft.

Example `Vector<double> myV(5 , 2);`

Usage: `Vector<complex> myFFT=fftshift(fft(myV));`

`//this should be the same as 'myV'`

`Vector<double> myIFFT=Re(ifft(ifftshift(myV)));`

Vector::Math

Function:

- `Type min(Vector<T> &rhs)`
- `Type max(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- finds the max (or min) value inside the coord

Example Usage:

- `double mymin=min(oldC)`

Vector::Math

Function: **Type** `min(Vector<T1> &rhs, T2 num)`
 Type `max(Vector<T1> &rhs, T2 num)`

Input: rhs-->A Vector of the Type T1
num--> a number of type T2

Description: eturns the min (or max) entry in the list in rhs and then the smallest (or largest) between 'num' and that number

Example `Vector c1(6);`

Usage: `double mymin=min(c1, 0.2); //the result here is 0.2`
`double mymin=min(0.2, c1/100); //this is also valid with the result of 0.06`

Vector::Math

Function:

- **Type dot(Vector<T> &c1 Vector<T2> &c2)**
Type norm(Vector<T> &rhs)

Input:

- rhs-->Another Vector of the SAME Type

Description:

- norm returns (if size()==3) $(x^*x + y^*y + z^*z)^{(1/2)}$
dot returns $(x1*x2+y1*y2+z1*z2)$

Example Usage:

- `double mynorm=norm(oldC);
double dd=dot(oldC, newC);`

Vector::Math

Function:

- `Vector<T> sin(Vector<T> &rhs)`
- `Vector<T> cos(Vector<T> &rhs)`
- `Vector<T> tan(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- performs an element-by-element evalutation of the trig functions sin, cos, tan

Example Usage:

- `newC=cos(oldC*Pi/180);`

Vector::Math

- Function:**
- `Vector<T> sinh(Vector<T> &rhs)`
 - `Vector<T> cosh(Vector<T> &rhs)`
 - `Vector<T> tanh(Vector<T> &rhs)`
- Input:**
- `Vector<T>`
- Description:**
- performs an element-by-element evalutation of the hyperbolic trig functions sinh, cosh, tanh
- Example Usage:**
- `newC=tanh(oldC*Pi/180)`

Vector::Math

Function:

- `Vector<T> sqrt(Vector<T> &rhs)`
- `Vector<T> sqr(Vector<T> &rhs)`

Input:

- `Vector<T>`

Description:

- element-by-element square and square root

Example Usage:

- `newC=sqr(oldC)`

Vector::Math

Function:

- **Type sum(Vector<Type> &c1)**
- c1-->A Vector of the numerical type 'Type'
- takes the sum af a Vector v[0]+v[1]+...v[N]
- `Vector<double> v1(3,4)`
`double dd=sum(v1); //result is 12`

Input:

Description:

Example Usage:

Vector::other

Function:

- `template<class Func>`
`void apply(Func &f)`

- `template<class Func>`
`void apply(Func &f, const Range &r)`

Input:

- `f-->` a class with the `T` operator()`(T in, int i)` defined for it.

- `r-->` the row range object

Description:

- applies a function on the data inside the matrix...the class 'Func' MUST have this operator

`T operator()(T in, int i)`

where 'in' is the vector data element, and `i` is the index defined in order for the apply to work.

If the Ranges are specified then only those elements in the vector specified in the ranges will be effected.

If no ranges are specified all the elements will be effected.

Example

```
□ class myF{  
myF(){}  
double T operator()(double in, int i){  
    return (i+1);  
}  
  
Vector<double> fm(2,5); //fm={5,5}  
myF func;  
fm.apply(func); //fm={1,2}
```

Usage:

Vector::other

Function:	<input type="checkbox"/> bool empty()
Input:	<input type="checkbox"/> none
Description:	<input type="checkbox"/> returns true if length==0 flase if not
Example Usage:	<input type="checkbox"/> <code>Vector<complex> tmpVec;</code> <code>bool l=tmpVec.empty(); //would be true</code>

Vector::other

Function:

□ **Vector<Type> fill(T &rhs)**

Input:

□ rhs->a number

Description:

□ fills the vector with the value of the rhs

Example Usage:

□ `Vector<> moo(3);
moo.fill(4); //moo=[4 , 4 , 4];`

Vector::other

Function: **void pop()**

Input: none

Description: resizes the vector to a length length-1...
it preserves all the data and just dropping the LAST element in the list
this is equivalent to 'resizeAndPreserve(size()-1)'

Example `Vector<complex> tmpVec(3, 2);`

Usage: `tmpVec.pop(); //new vector length is 2 with elements 1--2 are all still 2`

Vector::other

Function: `void push_back(T num)`

Input: `num` --> Adds the element 'num' to the back of the list and increases the vector length by one

Description: `resizes the vector to a length length+1...`
`this is like 'resizeAndPreserve(size()+1, num)'`

Example `Vector<complex> tmpVec(3, 2);`

Usage: `tmpVec.push_back(complex(0,3)); //new vector length is 4 with elements 1--3 are all still 2 the 4th elemenet is complex(0,3)`

Vector::other

- Function:**
- `void resize(int newL)`
 - `void resize(int newL, T num)`
- Input:**
- `newL` --> the new size of the vector
 - `num` --> the vector will be filled with 'num' after resizing
- Description:**
- resizes the vector to a length `newL...`
- Example Usage:**
- `Vector<complex> tmpVec(3, 2);`
 - `tmpVec.resize(5); //new vector length is 5`
 - `tmpVec.resize(5, 4); //new vector length is 5 filled with 4's`

Vector::other

Function: `void resizeAndPreserve(int newL)`
`void resizeAndPreserve(int newL, T num)`

Input: newL --> the new size of the vector

Description: resizes the vector to a length newL...
if newL < old length then ALL the data up to newL is kept...
if newL > old Length then all data is kept, but the NEW entries are NOT initialized (or are filled with 'num')

Example `Vector<complex> tmpVec(3, 2);`

Usage: `tmpVec.resizeAndPreserve(5); //new vector length is 5 with elements 1--3 are all still 2`
`tmpVec.resizeAndPreserve(5, complex(0,3)); //new vector length is 5 with elements 1--3 are all still 2 and 4-5 are complex(0,3)`

Vector::other

Function: `■ Vector<T> rotateRight(Vector<T>, int n)`

Input: `■ Vector<T>` --> a vector of any type

`n` --> the amount of points to shift the vector

Description: `■` Rotates a vector by placing the last 'n' elements at the begining and the first 'size-n' elements at the end. 'n' should be NO larger then size()/2

Example `■ Vector<double> myV(6);`

Usage: `//same as 'fftshift'`

`myV=rotateRight(myV, 3);`

Vector::other

Function:

- `int size()`
- `int length()`

Input:

- `none`

Description:

- returns the length of the Vector

Example Usage:

- `Vector<complex> tmpVec(3, 2);`
- `int l=tmpVec.size();`

--Global Constants	template<class Type, class Structure> class _matrix : protected MemChunkRef<Type>{....
--Constructors	Accepted Template Params <ul style="list-style-type: none"> o Type --> another class...typically a number type (int, double, complex, float....)...but can be anything (NO default) o Structure --> The Matrix form (FullMatrix, HermitianMatrix, SymmetricMatrix, DiagonalMatrix, IdentityMatrix)
--matrix	MemChunk is a class that handles the memory allocation and deallocation for ALL the various containers here
--matrix	
--Element Extraction	
--col	
--data	
--get	there are 2 classes of mathematical operations the first ELEMENT BY ELEMENT and then the 'total' operations (those that require the use of temporaries and total matrix access)
--getCol	
--getRow	
--operator()	
--operator(range,range)	
--row	
--Assignments	
--copy	
--operator=	
--put	
--putCol	
--putRow	
--Math	
--*_=	
--+,-,/*	
--+,-=, *=, /=	
--==, !=	
--chop	
--conj	
--cross(vec,vec)	
--cross, tensor_product	
--diag	
--exp,log	
--floor, ceil, abs	
--GramSchmidt	
OrthoNorm	
--inv	
--LU	
--LUbackSub	
--LUdecomp	
--LUsolve	
--matrix*vector	
--Mexp, Mlog	
	<ul style="list-style-type: none"> • FullMatrix <ul style="list-style-type: none"> o This is the most general matrix. It assumes each element is unique. IT also requires the most storage (rows*cols elements) • HermitianMatrix <ul style="list-style-type: none"> o This should always be of the 'complex' number type. Only the diagonal and the upper triangle of the matrix is stored. The Lower triangle is assumed to be the conjugate of the upper triangle. The elements Along the DIAGONAL ARE FORCED TO BE REAL! • SymmetricMatrix <ul style="list-style-type: none"> o This is like the Hermitian Type except that ALL the elements are REAL and so the upper triangle IS the lower triangle • DiagonalMatrix <ul style="list-style-type: none"> o Contains ONLY elements along the diagonal • IdentityMatrix <ul style="list-style-type: none"> o Contains only ONE element the number '1' but is treated like a DiagonalMatrix in terms of 1's all along the diagonal
	<ul style="list-style-type: none"> • To use matrix include the file "matrix.h" this contains the typically used matrix types as typdefs to avoid lots of typing <ul style="list-style-type: none"> • matrix --> _matrix(complex, FullMatrix) • rmatrix --> _matrix(double, FullMatrix) • hmatrix --> _matrix(complex, HermitianMatrix) • smatrix --> _matrix(double, SymmetricMatrix)

--- prop, adjprop
--- QR
--- Re, Im
--- sin, cos, tan,
asin, acos, atan,
sinh, cosh, tanh,
asinh, acosh, atanh
--- trace
--- transpose, adjoint
--IO
--- operator<<
--Other Functions
--- apply
--- cols
--- elements
--- empty
--- fill
--- identity
--- position
--- reshape
--- resize
--- resizeAndPreserve
--- rows
--- type

- **dmatrix** --> _matrix(complex, DiagonalMatrix)
 - **rdmatrix** --> _matrix(double, DiagonalMatrix)
 - **imatrix** --> _matrix(complex, IdentityMatrix)
 - **rimatrix** --> _matrix(double, IdentityMatrix)
-

!!NOTE!!

PLEASE SEE 'operator=' for IMPORTANT INFO

The default compiling options DOES NOT CHECK ANY BOUNDS. If you go outside the matrix sizes you will crash the program. This is for SPEED. If it does not have to check the bounds then it is one less operation it must do. YOU CAN TURN THIS BACK ON by defining 'MatDeBug 1' in the compile string
i.e. g++ --DMatDeBug myprog.cc

The default option DOES NOT CHECK ASSIGNMENTS!! (i.e. a identitymatrix=fullmatrix would be allowed). For correctness sake this should not be allowed as it would loose all the info in the full matrix. It is turned off because in most cases you know your matrix contents....and doing things like Hermitian=Full is okay especially when constructing Hamiltonians from FullMatrix Parts when you know that the end matrix will be Hermitian..

To Turn on the Checking either "#define MatAssignCheck 1" or
g++ -DMatAssignCheck myfile.cc

matrix::global constant

Function: `enum MatrixType{Mgeneral, Mfull, Mdiagonal, Midentity, Msymmetric, Mhermitian};`

Input:

Description: a data elements for the matrix types

Example

Usage:

matrix::constructor

Function:	<input type="checkbox"/> <code>_matrix<T, Stucture>()</code>
Input:	<input type="checkbox"/> None
Description:	<input type="checkbox"/> empty constructor, sets rows=0, cols=0
Example Usage:	<input type="checkbox"/> <code>rmatrix moo;</code>

matrix::constructor

Function: `_matrix(const _matrix<T1, S1> &cp)`

Input: cp --> another matrix that can be of different number type AND structure

Description: copy constructor

Example `rdmatrix mool;`

Usage: `matrix moo2(mool); //no info lost here`

`rimatrix moo3(mool); //info Will be LOST (i.e. diagonal-->identity)`

matrix::constructor

Function: `matrix(int rs, int cs)`

Input: `rs--> the number of rows for the new matrix`
`cs--> the number of cols for the new matrix`

Description: `Creates a new matrix WITH NO initialization`

Example Usage: `matrix moo2(3, 5); //a 3x5 complex full matrix`

matrix::constructor

Function: `matrix(int rs, int cs, T1 num)`

Input: `rs`--> the number of rows for the new matrix
`cs`--> the number of cols for the new matrix
`num`--> a number of Type 'T1'

Description: `Creates a new matrix with all elements fill with num...it performs the type conversion to 'Type' if necessary or possible`

Example `matrix moo2(3, 5, 89); //a 3x5 complex full matrix filled with complex(89,0)`

Usage:

matrix::constructor

Function: `template<class T1>`
`_matrix(int rows, int cols, T1 **in);`

Input: rows--> the number of rows for the new matrix
cols--> the number of cols for the new matrix
in--> a pointer to a matrix data chunk of type T1

Description: creates a matrix from a data pointer. Since it cannot check the length of the data pointer, it better be the same size as 'rows' and 'cols' are or the program will crash. It will attempt to convert 'T1' to 'T', and will create a memory copy of the data, not just reference the data.

Example `double data[2][2]={{1,1},{2,2}};`

Usage: `rmatrix myMa(2,2,data);`

matrix::constructor

Function:

□ `_matrix(int rows, int cols, T *in):`

Input:

□ rows--> the number of rows
cols-->the number of columns in the matrix
in--> a data ptr to a vector...

Description:

□ fills matrix from a 1D data pointer
meant really for coping the data array

Example Usage:

□ `double *data={1,2,3,4};
rmatrix mm(2,2,data); //mm={{1,2},{3,4}}`

matrix::element extraction

Function: `Vector<T> col(int which)`

Input: `which` --> the column that you wish to extract from the matrix

Description: `Returns a vector of length "rows()" of the values of the column 'which' in the matrix`

`effecting the elements in the vector DOES NOT effect the elements in the matrix`

Example `hmatrix hm(3, 3, complex(0,2));`

Usage: `Vector<complex> rs=im.col(1); //returns the vector
(complex(0,2),0,complex(0,-2))`

`//Note:: hermitian matrix cannot have complex diagonal elements
thus upon filling the matrix with complex(0,2)...the diagonal gets
set to the real part`

matrix::element extraction

Function:

`T *data();`

Input:

none

Description:

return the pointer to the array of 'T' that MemChunk managed

if you alter these values you alter the matrix values

Example Usage:

`double *vdat=oldC.data();`

matrix::element extraction

Function:

- `T &get(int r, int c);`
- `T get(int r, int c) const;`

Input:

- `r--> the row to access`
- `c--> the column to access`

Description:

- Extracts an element from the Matrix;
- can also be used for assignment

Example Usage:

- `rmatrix m1(4,4, 2);`
- `double ele=m1.get(0,3); //extraction ele=2`
- `oldC.get(3,1)=56; //assignment`

matrix::element extraction

Function: `□ Vector<T> getCol(int c);`

Input: `□ c--> the coulmn index`

Description: `□ gets a column out of the matrix`

Example Usage: `□ rmatrix fm(4,4,5);
Vector<double> loo=fm.getCol(1); //loo=[5,5,5]`

matrix::element extraction

Function: `□ Vector<T> getRow(int r)`

Input: `□ r--> the row`

Description: `□ gets a row from the matrix`

Example Usage: `□ rmatrix fm(4,4,5);
Vector<double> loo=fm.getRow(1); //loo=[5,5,5,5]`

matrix::element extraction

Function:

```
□ T &operator(int r, int c);  
T operator(int r, int c) const;
```

Input:

```
□ r--> the row to access  
c--> the column to access
```

Description:

```
□ --Extracts an element from the Matrix;  
--can also be used for assignment
```

Example Usage:

```
□ rmatrix m1(4,4, 2);  
double ele=m1(0,3); //extraction  
oldC(3,1)=56; //assignment
```

matrix::element extraction

Function:

```
□ Vector<T> operator()(const Range &r, int c)
Vector<T> operator()(int r, const Range &c)
_matrix operator()(const Range &r, const Range &c)
_matrix operator()(int br, int bc, int er, int ec)
```

Input:

□ r-> a Range or index
c-> a range of index

br--> begining row index,
bc--> begining column index,
er--> ending row index
ec--> endgin column index

Description:

□ get sub elements of the matrix, either get
column subelements (Range, column)
row subelements (row, Range)
or
submatrices (Range, Range)

Example Usage:

```
□ rmatrix fm(5,5,2);
cout<<fm(Range(0,3), 4); // [5 5 5]
cout<<fm(Range(0,2),Range(0,2)); // { {5,5}, {5.5} }
```

matrix::element extraction

Function: `□ Vector<T> row(int which);`

Input: `□ which --> the row that you wish to extract from the matrix`

Description: `□ Returns a vector of length "cols()" of the values of the row 'which' in the matrix`

`Effecting the elements in the vector DOES NOT effect the elements in the matrix`

Example Usage: `□ rimatrix im(3, 3);
Vector<double> rs=im.row(2); //returns the vector (0,0,1)
rs(2)=5; //this only effects the Vecotr NOT the matrix`

matrix::assignments

Function: `void copy(const matrix<T, Structure> &rhs)`

Input: `rhs` --> a matrix as the same type as the object.

Description: `This performs a memory copy of the rhs into the matrix. This Can ONLY work for the SAME 'T' and 'SAME' Structure matrices. It is faster then the A=rhs operator because the 'memcpy' function is used. The 'T' in the matrix should also ONLY be a simple data type (like double, complex, etc). This is why the memcpy function is not used in general...`

Example `Random<> ryR(-2, 2);`

Usage: `rmatrix loo(5,5);`

`loo.apply(ryR);`

`rmatrix koo(5,5);`

`koo.copy(loo); //faster than koo=loo`

matrix::assignments

Function: `_matrix operator=(const _matrix<T1, S1> &rhs)`
 `_matrix operator=(const T &rhs)`

Input: rhs-->a matrix object OF THE SAME SIZE as left-hand-side(can be of different type)...the Structure can be different if the macro definition MatAssignCheck is undefined
rhs--> or a 'constant' number

Description: Assignment operator...does do Type conversions

Example `_matrix<float, DiagonalMatrix> dm(4,4,6);`

Usage: `rmatrix fm(4,4,7);`

`dm=fm; //if 'MatAssignCheck' is off this will compile but you will end up with a diagonal matrix filled with 7's accross the diagonal`
`fm=dm; //this is a valid assignment fm will still be a fullmatrix with only 6's along the diagonal`

`fm=complex(2,3); // a full matrix completely filled with the complex number complex(2,3);`

matrix::assignments

Function: □ `template<class T2>`

```
void put(int whereR, int whereC, const T2 num)
void put(Range whereR, int whereC, const Vector<T2> num)
void put(int whereR, Range whereC, const Vector<T2> num)
void put(Range whereR, Range whereC, const matrix num)
```

Input: □ whereR --> the row of the matrix (or range of rows)
whereC --> the column of the matrix (or range of columns)
num-->an number of Type 'T2', or Vector having the same length as the Range object, or a matrix having same length as the Ranges

Description: □ puts the element 'num' in the position '(whereR,whereC)'

Will only put the number 'num' in the matrix IF that matrix Type has that (row,col) allowed

Example □ `dmatrix dm(2,2);`

Usage: `dm.put(0,0,4); //places a 4 in the (0,0) element`

```
dm.put(1,0, 5); //DOES NOT PLACE ANYTHING a diagonalmatrix has not
off diagonal bits
```

```
//The vector should be as long as the range is
Vector<double> moo(2); moo[0]=1; moo[1]=5;
rmatrix fm(3,3,0);
fm.put(Range(1,3), 1, moo);// fm={{0,0,0},{0,1,0},{0,5,0}}
fm.put(1,Range(1,3), moo);// fm={{0,0,0},{0,1,5},{0,5,0}}
```

matrix::assignments

Function:

- `template<class newT>`
 `void putCol(int c, const Vector<newT> &in)`

- `template<class newT, int N>`
 `void putCol(int c, const coord<newT, N> &in)`

Input:

- `c-->` the column
- `in-->` the data vector to place in the matrix

Description:

- put an entire column from either a Vector (should be the same length as the number of columns) or from a coord<T,N> (the number of columns of the matrix should be N).

Example

```
□ rmatrix fm(3,3,0);  
Usage:     Vector<double> moo(3,0); moo[0]=1; moo[1]=3; moo[2]=50;  
         fm.putRow(1,moo); //fm={{0,1,0},{0,3,0},{0,0,50}}  
         coord<> goo(4,5,60);  
         fm.putCol(1,goo); //fm={{0,4,0},{0,5,0},{0,60,0}}
```

matrix::assignments

Function:

- `template<class newT>`
 `void putRow(int r, const Vector<newT> &in)`

- `template<class newT, int N>`
 `void putRow(int r, const coord<newT, N> &in)`

Input:

- `r`--> the row
- `in`--> the data vector to place in the matrix

Description:

- put an entire row from either a Vector (should be the same length as the number of columns) or from a coord<T,N> (the number of columns of the matrix should be N)

Example

```
rmatrix fm(3,3,0);  
Vector<double> moo(3,0); moo[0]=1; moo[1]=3; moo[2]=50;  
fm.putRow(1,moo); //fm={{0,0,0},{1,3,50},{0,0,0}}  
coord<> goo(4,5,60);  
fm.putRow(1,goo); //fm={{0,0,0},{4,5,60},{0,0,0}}
```

Usage:

matrix::Math

Function: `rmatrix operator*=(const rmatrix &A);`
`matrix operator*=(const matrix &A);`

Input: A--> a real or complex Full Matrix

Description: This is a special multiplication operator. It DOES NOT perform this operation ($B=B^*A$) .. IT DOES perform the DYSON TIME MULTIPLICATION... $U=A^*U$...

NOTE the order change from a 'normal' *= operator.

you can only use this for Real Full Matrices and Complex Full Matrices. This is faster then writing it out by hand (e.g. $U=A^*U$) because we save a few temporary copying steps.

Example `rmatrix A(4,4), U(4,4);`

Usage: `U.identity(); //make it the identity matrix`
`//fill it up`
`U*=A; //performs $U=A^*U$;`

matrix::Math

Function:

- `_matrix<UpT, UpS> operator+(const _matrx<T1, S1> &lhs, const _matrix<T2, S2> &rhs)`
- `_matrix<UpT, UpS> operator+(const _matrx<T1, S1> &lhs, const T2 &rhs)`
- `... similar for other operators...`

Input:

- lhs-->a matrix or number
- rhs-->a matrix or number

Description:

- Adds, Subs, Divides, multiplies rhs to the Left hand side

if the operator is *, or / then these are special matrix operations...

matrix/matrix--> means A*inverse(B)

matrix*matrix --> means MatrixMultiply NOT an element by element operation

Example

```
□ matrix rm(5, 5,6);  
    rimatrix im(5, 5);  
    hmatrix hm(5, 5,45);  
    rm=im+hm; // no data is lost here (UpS=Hermitian, UpT=complex)  
    hm=rm+im; //data is lost from the 'rm; (UpS=Full, UpT=complex)
```

Usage:

matrix::Math

Function:

```
■ _matrix operator+=(const _matrix<T2,S2> &rhs)  
■ _matrix operator+=(T2 &rhs)  
  
..similar for -=, /=, *=
```

Input:

□ rhs-->Another a matrix of number type 'T2' and strucutre "S2" OR a number

Description:

□ self math operation with a matrix or a number

matrix/=matrix--> means $A = A * \text{inverse}(B)$

matrix*=matrix --> means "Matrix Multiply" NOT an element by element operation

Example Usage:

```
■ rdmatrix dm(3, 3, 1);  
■ rmatrix fm(3, 3, 4);  
■ dm+=fm*2+4; //you will loose the info from fm here  
■ fm+=dm*5; //a proper matrix-matrix operations  
■ fm+=7; //add 7 to each element
```

matrix::Math

Function: **bool operator==(_matrix<T1,S1> &lhs, _matrix<T2, S2> &rhs)**

Input: lhs--> a matrix of number type 'T1' and structure 'S1'
rhs --> a matrix of number type 'T2' and structure 'S2'

Description: equality test "=="
if ALL the elements are the same this is true
in-equality test "!="
if ALL the elements are the same this is false

Example Usage: `bool test= m1==m2;`

matrix::Math

Function: **void chop(double eps)**
matrix chop(matrix &in)

Input: rhs-->A matrix of any type
eps--> the largest allowed number allowed in the matrix.

Description: sets all entries whos ABSOLUTE VALUE is less then 'eps' to 0.

THe external function simply chops everything BELOW 1e-12 to 0...it is not possible to set the 'eps' value for this function.

Example `matrix cc(4,4, 1.e-8);`

Usage: `cc.chop(1.e-7); //all elements are set to zero`
`//OR--`
`matrix moo=chop(cc);`

matrix::Math

Function: `matrix conj(matrix &rhs)`

Input: `rhs`--> A matrix of any type

Description: `rhs` takes the conjugate of each element in a matrix

Simply switches the sign of the complex part of any element (if the matrix is not complex this will still work, it will just do nothing)

Example `hmatrix hm(4,4, complex(3,4));`

Usage: `hmatrix hm2=conj(hm); // hm2 has the upper triangle now all complex(3,-4) and the lower diagonal all complex(3,4)`

matrix::Math

Function: `_matrix<UpT, FullMatrix> cross(const Vector<T1> &rhs, const Vector<T2> &lhs)`

Input: rhs-->A Vector of any type
lhs--> A Vector of any type (its length should be the same as rhs)

Description: cross product between 2 vectors...does do Type conversions to UpT
returns a FULL matrix of Length x Length where length is the length of the vector(s)

Example `Vector<complex> vm(5 , 5);`

Usage: `Vector<complex> vm2(5 , 2);`
`matrix fm=cross(vm, vm2);`

matrix::Math

Function: `_matrix<UpT, UpS> cross(_matrix<T1, S1> &A, _matrix<T2, S2> &B)`
 `_matrix<UpT, UpS> tensor_product(_matrix<T1, S1> &A, _matrix<T2, S2> &B)`

Input: A--> a matrix
B--> a matrix

Description: returns the 'tensor product', 'cross' product, or 'kroneker product' of two matrices
the return matrix will be of the size "lhs.rows()*rhs.rows()"x"lhs.cols()*rhs.cols()"
The UpS will be determined as the largest of S1 or S2
the UpT will be determined as the largest of T1 and T2

Example `matrix m1(4,4,3);`
Usage: `matrix m2(5,5,2);`
`matrix m3=cross(m1, m2); //m3 is a 20x20 matrix`
`m3=tensor_product(m1,m2); //Same as 'cross'`

matrix::Math

Function: `void diag(_matrix &rhs, _matrix<Type, DiagonalMatrix> &eigenvalues, _matrix<Type, FullMatrix> &eigenvectors)`

Input: rhs-->any matrix you wish to diagonalize

eigenvalues--> A Diagonal matrix that will contain the eigenvalues

eigenvectors--> a Full matrix that will contain the eigenvectors of the system

Description: Solves the General Eigenvalue problem...it is smart about which method it chooses to diagonalize the matrix

if rhs-->HermitianMatrix-->uses a complex Householder algorithm

if rhs-->Symmetric--> uses a real Householder algorithm

if rhs-->Diagonal or Identity--> no need to diagonalize

if rhs-->Full--> uses a QR type method

Example `matrix MyMat(4,4,6);`

Usage: `dmatrix diags(4,4);`

`matrix evect(4,4);`

`diag(MyMat, diags, evect); //uses the QR type method for
diagonalization`

matrix::Math

Function: `_matrix exp(_matrix &rhs);`
 `_matrix log(_matrix &rhs)`

Input: rhs-->a matrix

Description: takes the exp or log of each ELEMENT in the matrix (NOT a matrix Exp or Matrix Log...see Mexp and Mlog)

Example `newC=exp(oldC);`
Usage:

matrix::Math

Function:

- `_matrix floor(_matrix&rhs);`
- `_matrix ceil(_matrix&rhs);`
- `_matrix abs(_matrix&rhs)`

Input:

- `rhs->a matrix`

Description:

- takes the floor, ceil, or abs of each element in the matrix

Example Usage:

- `newC=ceil(oldC)`
- `newC=abs(oldC)`
- `newC=max(oldC)`

matrix::Math

Function: `matrix GramSchmidt(matrix &rhs)`
`matrix OrthoNorm(matrix &rhs)`
`matrix GramSchmidt(matrix &rhs, Vector<double> &norms)`

Input: `rhs-->any matrix`
`norms--> a vector that will contain the NORMALIZATION constants necessary to perform the orthonormalization...these norms can be useful in other computations....`

Description: `performs the Gram Schmidt Orthonormalization to COLUMNS of the matrix...Errors will occur for a Singular matrix...the columns are NORMALIZED and ORTHOGONAL after the process`

Matrix MUST BE SQUARE

The normalization constants are held in "norms"

Example `matrix MyMat(4,4,6);`
Usage: `matrix MatN=GramSchmidt(MyMat); //matrix orthonormalization`
`Vector<double> n;`
`MatN=GramSchmidt(MyMat,n); //matrix orthonormalization with norms`
`contained in 'n'`

matrix::Math

Function: `matrix<Type, invers_structure> inv(_matrix &rhs);
void inv()`

Input: `rhs-->` a matrix

Description: `Matrix Inverse`

Solves the set of linear equations where $A x = b$ where b is looped through $(1,0,0,\dots)$, $(0,1,0,0,\dots)$, $(0,0,1,0,\dots)\dots(0,0,\dots,0,1)$ as shown in LUBackSub...

the result is the inverse of the rhs matrix

Matrix MUST BE SQUARE

both the 'external' function and 'internal' function are provided

Example `matrix MyMat(4,4,6);`
Usage: `matrix MatInv=inv(MyMat); //valid matrix inverse
//or you can use the internal function
MayInv=MyMat.inv(); //also a valid for the matrix inverse`

matrix::Math

Function: `int LU()`

```
int LU(_matrix<T, FullMatrix> &LU);  
int LU(_matrix<T, FullMatrix> &L, _matrix<T, FullMatrix> &U);
```

Input: LU--> a full matrix

U, L--> full matrix

Description: destructive LUdecomp's

```
int LU()
```

non destructive LU's

...(slower, but potential more useful) where the LU matrices are either stored in compact form (LU(matrix)) or in separate matrices (LU(L,U))

will return '-1' if the matrix is singular and '1' if not

Example `matrix fm(4,4,3), L, U;`

Usage: `fm.LU(L,U); //non-destructive LU decomposition`

matrix::Math

Function: `void LUbackSub(Vector<int> &rowperm, Vector<T> &b)`

Input: `rowperm`--> the permutation of the matrix from an LUdecomp

`b`--> a vector to solve the linear set of equations $A x = b$ ('A' is the matrix)

The Solutions will be IN 'b'

Description: `Solve the linear system 'A x=b' with a variety of different b's assuming that you have already performed an 'LUdecomp' on the matrix...the matrix is preserved...if you wish to reuse the decomposition matrix perform the solve in this order`

this is used as a functional step in either creating the matrix Inverse or for solving a linear system of equations

Example `Vector<double> b=...;`

Usage: `matrix MyMat=...;`

`matrix LUD=MyMat;`

`Vector<int> rp(MyMat.rows(), 0);`

`LUD.LUdecomp(rp);`

`for(...){`

`Vector<double> mysol=b;`

`LUD.LUBackSub(rp,mysol);`

`...do something with the solutions...`

`...change 'b'...`

`}`

matrix::Math

Function: **int LUdecomp(Vector<int> &rowperm)**

Input: rowperm-->a storage place for the row permutations (should be of length "rows()")

Description: LU decomposition of the matrix (does this inplace returning the row permutation vector)

This is a destructive decompositon (the matrix will be replaced with the LU matrix)

The 2 matrices "L" and "U" are contained inside the single matrix

The Matrix MUST be square

this is used as a functional step in either creating the matrix Inverse or for solving a linear system of equations

The return value will be '1' if the matrix is NOT singular, but '-1' if the matrix is singular. An error message will be posted to cerr if the "LUwarn" flag is true, to turn the warning off set "LUwarn=false"

if the

Example `matrix fm(4,4,3);
Vector<int> rp(4,0);
fm.LUdecomp(rp);`

Usage:

matrix::Math

Function: **void LUsolve(Vector<T> &b)**

Input: b--> a vector to solve the linear set of equations A x= b ('A' is the matrix)

Description: DESTRUCTIVE..b will be overwritten with the solutions, and the matrix will be the LU decomposition in compact form...

Solve the linear system 'A x=b' ONLY ONCE...the matrix afterwards will be GARBAGE... if you wish to reuse the decompositon matrix perform the solve in this order

Example //To perform a Solve where you can reuse the matrix Do THIS

Usage: Vector<double> b=...;

matrix MyMat=...;

matrix LUD=MyMat;

Vector<int> rp(MyMat.rows(), 0);

LUD.LUdecomp(rp);

for(...){

Vector<double> mysol=b;

LUD.LUBackSub(rp,mysol);

...do something with the solutions...

...change 'b'...

}

matrix::Math

Function:

- `Vector<UpT> operator*(const _matrix<T1, S1> &rhs, const Vector &lhs)`

- `Vector<coord<T, N> > operator*(const _matrix<T1, S1> &rhs, const Vector<coord<T, N> >&lhs)`

Input:

- rhs-->A matrix of anytype
- lhs--> A Vector of any type (its length should be the same number of rows as the matrix)

Description:

- Multiplication operator...does do Type conversions to UpT

for matrix*Vector<coord<>>
!!if the Vector is a Vector<coord<T,N>> then the number of columns in the Matrix should be
N*size(vector). It treats the coord vector as a fully unrolled object !!

Example

- `_matrix<float, DiagonalMatrix> dm(4, 4, 6);`

Usage:

- `Vector<complex> vm(dm.rows, 5);`
- `Vector<complex> mulV=dm*vm;`

matrix::Math

Function: `_matrix<Type, OutStruct> Mexp(_matrix &rhs, T number=1);`
`_matrix<Type, OutStruct> Mlog(_matrix &rhs, T number=1);`

Input: `rhs->matrix`

Description: The Matrix Exponential calculated by diagonalizing the lhs, and recollecting the parts as

$$\text{adjoint}(U) * \exp(\text{diagonal} * \text{number}) * U$$

The Matrix Logarithm calculated by diagonalizing the lhs, and recollecting the parts as

$$\text{adjoint}(U) * \log(\text{diagonal} * \text{number}) * U$$

--If you wish to perform this operation `exp(M*i t)` TO GET CORRECT RESULTS AND FASTER PERFORMANCE USE "Mexp(Matrix, i*t)" this correctness issue arrises from the casting of matrix type as descrirbed in the 'operator='

Example `matrix fm(4, 4, 3);`

Usage: `fm=Mexp(fm, complex(0, 0.2)); //standard QM Spin propogator`
`fm*=complex(0,0.2);`
`fm=Mexp(fm); //this is slower then the method on the line above`
`(and sometimes can produce ODD results...moral--> USE THIS FOR PROPOGATORS!)`

matrix::Math

Function: `_matrix<UpT, UpS> prop(_matrix<T1, S1> &U, _matrix<T2, S2> &ro)`
`_matrix<UpT, UpS> adjprop(_matrix<T1, S1> &U, _matrix<T2, S2> &ro)`

Input: U->matrix
ro->matrix

Description: "prop" performs this operation

$U^*ro^*\text{adjoint}(U)$

~30% more efficiently then the explicit version above

"adjprop" performs this operation

$\text{adjoint}(U)^*ro^*U$

~30% more efficiently then the explicit version above

Example `hmatrix rm(3,3,2);`

Usage: `dmatrix dm(3,3);`
`matrix evects(3,3);`
`diag(rm, dm, evects); //evects will be unitary because we are`
`diagonalizing a hermitian matrix`
`hmatrix rm2=prop(evects, rm);`

matrix::Math

Function:

□ **void QR(_matrix &rhs, matrix &Q, matrix &R)**

Input:

□ rhs-->the matrix input matrix you wish to perform the QR upon
Q--> the 'Q' matrix that will be filled with the proper values
R--> The 'R' matrix that will be filled with the upper triangular values..

Description:

□ performs the QR algorithm to the rhs of the matrix

Example Usage:

□ `matrix MyMat(4,4,6), Q(4,4,0), R(4,4,0);
QR(MyMat, Q, R); //a simple QR decomposition...`

matrix::Math

- Function:** `_matrix Re(_matrix &rhs);`
`_matrix Im(_matrix &rhs);`
- Input:** rhs--> a matrix
- Description:** takes the Real part(Re) or Imaginary part (Im) of each element of a matrix.
if the matrix is Real then "Im" does nothing, and will return a matrix of zeros
- Example Usage:** `hmatrix hm(4,4, complex(3,4));`
`hm.put(2,1, complex(0,8));`
`hmatrix hm2=Re(hm); //all the complex parts are now 0`
`rmatrix fm(5,5,3);`
`fm=Im(fm); //fm is now a bunch of zeros`

matrix::Math

Function: `_matrix sin(_matrix &rhs)`
...similar for other trig functions...

Input: rhs->a matrix

Description: performs the various trig functions to EACH ELEMENT. These are not Matrix Trig function. To perform those operations you will need to do a diagonalization

Example `newC=tanh(oldC*Pi/180)`
Usage:

matrix::Math

Function:	<code>SumType(T) trace(_matrix &A);</code> <code>SumType(UpT) trace(_matrix<T1,S1> &A, _matrix<T2, S2> &B)</code>
Input:	<ul style="list-style-type: none">□ A-->a matrix□ B-->a matrix
Description:	<ul style="list-style-type: none">□ the single matrix trace returns $\text{Sum}(A(i,i))$the double matrix trace performs a highly optimized version of $\text{trace}(A^*B)$there is no need to perform the A^*B multiplication if only the trace is desired
Example Usage:	<ul style="list-style-type: none">□ <code>rmatrix rm(3,3,2);</code>□ <code>double tr=trace(rm); //tr=6</code>□ <code>rmatrix rm2(3,3,4);</code>□ <code>tr=trace(rm, rm2);</code>

matrix::Math

Function: `_matrix transpose(_matrix &rhs);`
`_matrix adjoint(_matrix &rhs)`

Input: rhs--> a matrix

Description: transpose takes the transpose of a matrix
Simply switches the (i,j) indexing to (j,i) and is thus an element by element type operator

adjoint takes the conjugate transpose of a matrix

Simply switches the (i,j) indexing to (j,i) and is thus an element by element type operator AND takes the conjugate of the element

Example `hmatrix hm(4,4, complex(3,4));`

Usage: `hm.put(2,1, complex(0,8));`

`hmatrix hm2=adjoint(hm); //the same as conj(transpose(hm))`

`hmatrix hm3=transpose(hm); //now the lower triangle of the matrix
is the positive side`

matrix::IO

Function: `std::ostream &operator<<(std::ostream &otr, _matrix<complex, structure> oo);`

Input: `otr-->` an output stream

`oo-->`the matrix you wish to ouput

Description: `prints out a matrix that is readable directly by matlab in the format`

```
<MatrixType> <rows>x<Cols>
[ [ ... ]
[ ... ]
...
]
```

Example `rmatrix b(2,2,4);`

Usage: `cout<<b;`

```
//prints
//FullMatrix 2x2
//[ [ 2 2]
//[2 2]
//]
```

```
matrix cm(2,2,2);
cout<<cm;
//prints
// FullMatrix 2x2
// [ [ complex(2,0) complex(2,0) ]
// [ complex(2,0) complex(2,0) ]
// ]
```

matrix::other

Function:

- `template<class Func>`
`void apply(Func &f)`

- `template<class Func>`
`void apply(Func &f, const Range &r, const Range &c)`

Input:

- `f-->` a class with the `T operator()`(`T in, int i, int j`) defined for it.

`r-->` the row range object

`c-->` the column range object

Description:

- applies a funciton on the data inside the matrix...the class 'Func' MUST have this operator

`T operator()(T in, int i, int j)`

where 'in' is the matrix data element, and `i,j` are th row, column indexes defined in order for the apply to work.

If the Ranges are specified then only those elements in the matrix specified in the ranges will be effected.

If no ranges are specified all the element will be effected

Example

```
□ class myF{  
myF(){ }  
double T operator()(double in, int i, int j){  
return (i+1)*(j+1);  
}  
  
rmatrix fm(2,2,5); //fm={{5,5},{5,5}}  
myF func;  
fm.apply(func); //fm={{1,2},{2,4}}
```

Usage:

matrix::other

Function:	<ul style="list-style-type: none">□ <code>int cols()</code>
Input:	<ul style="list-style-type: none">□ none
Description:	<ul style="list-style-type: none">□ returns the number of columns in the matrix
Example Usage:	<ul style="list-style-type: none">□ <code>rimatrix im(3, 3);</code> <code>int rs=im.cols(); //returns the number '3'</code>

matrix::other

Function:

- int elements()**

Input:

- none

Description:

- returns the number of VALID elements in the matrix
 - is rows*cols for fullmatrix
 - is rows*(rows+1)/2; for hermetian and symmetric matrix
 - is rows for diagonal matrix
 - is 1 for identity matrix

Example Usage:

- `rimatrix im(3, 3);
int rss=im.elements(); //is '1'`

matrix::other

Function: `bool empty()`

Input: `none`

Description: `returns true if rows==0 and cols==0 false if not`

Example Usage:

```
matrix fm;
bool l=fm.empty(); //would be true
rmatrix mm(1,1);
bool l=mm.empty(); //would be false
```

matrix::other

Function: **template<class T2>**
 void fill(T2 Num)

Input: Num--> fills every valid entry in the matrix with 'Num'

Description: Sets every possible element to 'Num'

Example Usage: `rmatrix im(3, 3, 9); //a matrix full of 9's
im.fill(6); //now the matrix is full of 6's
imatrix km(4,4); // identity matrix
fm.fill(7); //NOT POSSIBLE identity matrix only has 1's....
dmatrix lm(2,2); //diagona matrix
lm.fill(56); //places 56 along the diagonal (but NOwhere else)`

matrix::other

Function: `void identity()`
`void identity(int newSize)`

Input: `newSize`--> will resize the matrix to `newSize x newSize`

Description: `Sets the matrix to the Identity Matrix (i.e. only ones along the diagonal) it DOES NOT change the matrix type...just sets things to 0 and 1`

`if "newSize" is preset`

`first resizes the matrix to 'newSize x newSize' then...`

`then sets the matrix to the Identity Matrix (i.e. only ones along the diagonal) it DOES NOT change the matrix type...just sets things to 0 and 1`

Example `rmatrix im(3, 3, 9); //a matrix full of 9's`

Usage: `im.identity(); //now the matrix has ones along the diagonal and 0's elsewhere`
`im.identity(6); //now the matrix has ones along the diagonal and 0's elsewhere and is no a 6x6 matrix but is still a full matrix`

matrix::other

Function: `int position(int r, int c)`

Input: `r--> the row of the matrix`
`c--> the column of the matrix`

Description: `returns the position in the 'data()' vector of the proper element in the list. The data for the matrix is actually stored as a Vector..the operators know which element is which in the vector using this function`

Example `rdmatrix dm(3, 3);`

Usage: `int pos=dm.position(1,1); //pos would be '2'`
`double secondele=dm.data()[pos]; //would be the same as 'dm(1,1)'`

matrix::other

Function: **void reshape(int newR, int newC);**

Input: newR--> the new number of rows
newC--> the new number of columns

Description: Simply redefines the size of the matrix without altering any of the elements. As a result newR*newC MUST BE the same as rows()*cols().

Example `matrix mat(4,4, 5);`

Usage:
`//a 2x8 matrix
mat.reshape(2,8);

//THIS WILL FAIL
mat.reshape(3,5);`

matrix::other

Function: **void resize(int newR, int newC)**
 void resize(int newR, int newC, T num)

Input: newR --> number of new rows
 newC --> number of new columns

Description: resizes the matrix to a size (newR, newC)...
if num is present the matrix will be filled with that object

Example Usage:

```
hmatrix hm;
hm.resize(5,5); // a 5x5 hermitian matrix
smatrix sm(4,4);
sm.resize(5,5, 4.0); //new matrix length is 5x5 filled with 4's
```

matrix::other

Function: **void resizeAndPreserve(int newR, int newC)**

Input: newR--> the new Row size

newC--> the new Column size

Description: resizes the matrix to a size (newR, newC) preserving all the data it can (i.e. if the resize is smaller than the current size, you will lose the over index values)

Example `rmatrix fm(5,5, 7);`

Usage: `fm.resizeAndPreserve(6,6);`

matrix::other

Function:	<ul style="list-style-type: none">□ <code>int rows()</code>
Input:	<ul style="list-style-type: none">□ none
Description:	<ul style="list-style-type: none">□ returns the number of rows in the matrix
Example Usage:	<ul style="list-style-type: none">□ <code>rimatrix im(3, 3);</code> <code>int rs=im.rows(); //returns the number '3'</code>

matrix::other

Function:	<input type="checkbox"/> MatrixType type()
Input:	<input type="checkbox"/> none
Description:	<input type="checkbox"/> Returns the enum 'MatrixType' of the matrix type
Example Usage:	<input type="checkbox"/> <code>rimatrix im(3, 3);</code> <code>MatrixType rs=im.type(); //is 'Midentity'</code>

--Global Constants template<class Engine_t>
--- iterator class Grid : public Engine_t{....

--Constructors

--- Grid
--- Grid

--Element Extraction

--- dim
--- dr
--- dx,dy,dz
--- min, max
--- operator()
--- Point
--- x(),y(),z()
--- x(t), y(t), z(t)
--- Xdim, Ydim, Zdim
--- Xmin,Ymin,Zmin,
Xmax, Ymax, Zmax

--Assignments

--- dim
--- Min, Max
--- operator=
--- Xdim,Ydim, Zdim
--- Xmin,Ymin,Zmin,
Xmax, Ymax, Zmax

--Other Functions

--- cellVolume
--- Center
--- Scale
--- size
--- Translate

Examples

--- looping 1
--- looping 2

Accepted Template Params

- **Engine_t** --> another class...THE main driver for the Grid...contains the shapes, points, (and basically everything about the Grid)
- The main 'Grid' class acts as a global container for all the various grid Engines...the functions described below have are identified by their Engine_t name

- Inside the Grid classes are hidden all sorts of optimizations to save both memory space AND speed. The memory space is managed by minimizing the amount of stored data points in the grid. For instance a Cube grid needs only ONE Vector of data going from min--max. The entire 3D cube can be expressed as combinations of elements from this list. The access to the points of the Grid is maintained by extraction operators.
- The **UniformGrid** is the most up-to-date and functional. The others are less so (because they are quite specialized for each shape furthermore the UniformGrid has been chosen as the Master Shape Grid Base Engine (see 'Shape')

Grid Engines...The Main Driver sets

- **NullGrid**
 - Nothing, has nothing, is nothing but a name and a place for empty sets
- **UniformGrid**
 - A Rectangular 3D grid. The uniform implies a constant step size between each point...each direction (x,y,z) can have a different step size
- **SurfCubeGrid**
 - Creates a cube in 3D that is NOT filled, only the surface is populated with points
- **FullCubeGrid**
 - creates a simple filled cube. All steps sizes are the same along each x,y,z direction
- **SurfSphereGrid**
 - Creates a Surface Sphere (NOT FILLED) in polar coordinates. (If you wish a sphere in Cartesian coords use the 'Shape' class)
- **FullSphereGrid**
 - Creates a Filled Sphere in polar coordinates. (If you wish a sphere in Cartesian coords use the 'Shape' class)

!!NOTE!!

See the 'Looping' Example for important Info.

grids::global constant

Function: `typedef Engine_tIter iterator`

Input:

Description: The Grid Iterator Type

Example Usage: `Grid<UniformGrid> myG;`
`Grid<UniformGrid>::iterator myIt(myG);`

grids::constructor

Function:

Grid()

none

Null Contructor

Example Usage:

`Grid<NullGrid> moo;`

grids::constructor

- Function:** `□ Grid(Grid &cp);`
- Input:** `□ cp--> a grid with the SAME ENGINE_T`
- Description:** `□ the copy constructor. It will copy the 'Engine_t' class as well.`
- Example Usage:** `□ Grid<SurfSphereGrid> sg(2,0, PI2, 0,PI2,100);
Grid<SurfSphereGrid> sg2(sg); //okay to copy
Grid<UniformGrid> ugl(sg); //WILL FAIL incompatable engines`

grids::constructor

Function: **Grid(Engine_t &cp)**

Input: cp--> an Engine_t object

Description: will create a Grid from a previously decalred Engine_t

Example Usage: `SurfSphereGrid myE(2,0, PI2, 0,PI2, 100);
Grid<SurfSphereGrid> sg(myE);
Grid<UniformGrid> ugl(myE); //WILL FAIL incompatable engines`

grids::constructor

Function: `Grid(coord<> &min, coord<> &max, coord<int,3> &dim)`

Input: `min--> the (x,y,z) minima`

`max--> the (x,y,z) maxima`

`dim--> number of divisions along each direction`

Description: `creates a Grid ASSUMING that the Engine_t can take the inputs (coord<> &min, coord<> &max, coord<int,3> &dim)`

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int, 3> dims(10,10,10);`

`Grid<UniformGrid> myG(mins, maxs, dims); //a ready to be used grid`

grids::constructor

Function: `Grid(double min, double max, int dim)`

Input:

- min-->a single minimum value
- max-->a single maximum value
- dim-->a single dimension

Description:

- creates a grid object assuming all 3 (x,y,z) directions have the same min, max, and dim

Example Usage:

- `Grid<SurfSphereGrid> sg(Pi/2, Pi/4, 10);`
- `Grid<UniformGrid> ug(1.,2.,3); //WILL FAIL`

grids::constructor

Function: `Grid(double xmax, double ymax, double zmax, int dim)`

Input:

- xmax--> maximum X (or R)
- ymax--> maximum Y (or theta)
- zmax--> maximum Z (or Phi)
- dim--> dimensions for each allowable direction

Description: ▫ A Sub constructor for a few Engines...simplifies some initialization

Example Usage: ▫ `Grid<SurfSphereGrid> sg(4.0,Pi/2, Pi/4, 10);`
`Grid<UniformGrid> ug(2.0, 3.0, 2.0, 100);`

grids::constructor

Function: `Grid(double xmin, double xmax, double ymin, double ymax, double zmin, double zmax, int dimx, int dimy, int dimz)`

Input:

- `xmin`--> minimum X (or R)
- `xmax`--> maximum X (or R)
- `ymin`--> minimum Y (or theta)
- `ymax`--> maximum Y (or theta)
- `zmin`--> minimum Z (or phi)
- `zmax`--> maximum Z (or Phi)
- `dimx`--> dimensions for the X direction
- `dimy`--> dimensions for the Y direction
- `dimz`--> dimentions for the Z direction

Description: A Sub constructor for a few Engines...simplifies some intialization

Example `Grid<SurfSphereGrid> sg(-1.0, 2.0, -2.0, 3.0, -2.0, 2.0, 10,`

Usage: `10,10); //WILL FAIL`

`Grid<UniformGrid> ug(-1.0, 2.0, -2.0, 3.0, -2.0, 2.0, 10, 10,10);`
`//GOOD`

grids::element extraction

Function:	<code>coord<> dim();</code>
Input:	<ul style="list-style-type: none">□ none
Description:	<ul style="list-style-type: none">□ returns the dimensions along all three directions
Example Usage:	<ul style="list-style-type: none">□ <code>coord<int> dims(5,5,5);</code> <code>Grid<UniformGrid> myG(mins, maxs, dims);</code> <code>myG.dim(); //will be [5,5,5]</code>

grids::element extraction

Function:

- `coord<> dr(); --> FOR ITERATORS`
- `coord<> dr(int i); --> FOR ENGINES`

Input:

- `i->` the point index

Description:

- returns the steps along each direction.

Example Usage:

```
□ coord<> mins(0,0,0), maxs(5,5,5);  
    coord<int> dims(5,5,5);  
    Grid<UniformGrid> myG(mins, maxs, dims);  
    myG.dr(0); //will be (1,1,1)
```

grids::element extraction

Function:

- `double dx(); --> FOR ITERATORS`
- `double dy(); --> FOR ITERATORS`
- `double dz(); --> FOR ITERATORS`

- `double dx(int i); --> FOR ENGINES`
- `double dy(int i); --> FOR ENGINES`
- `double dz(int i); --> FOR ENGINES`

Input:

- for iterators --> nothing
- i--> for engines..the index of the point

Description:

- returns the step size at point 'i'. Most likely this number is a constant for any direction.

Example Usage:

- `coord<> mins(-1, -2, -3), maxs(1,2,3);`
- `coord<int> dims(2,2,2);`
- `Grid<UniformGrid> myG(mins, maxs, dims);`
- `double dx=myG.dx(0); //will return 1.0`

grids::element extraction

Function:

- `coord<> min();`
- `coord<> max();`

Input:

- `none`

Description:

- returns the min or max along all 3 direction in a `coord<double, 3>`

Example Usage:

```
□ coord<> mins(-1, -2, -3), maxs(1,2,3);  
    Grid<UniformGrid> myG(mins, maxs, dims);  
    coord<> mins=myG.Min(); //will return (-1,-2,-3)
```

grids::element extraction

- Function:**
- `coord<> operator(int xd, int ys, int zd);`
- Input:**
- `xd-->element of the X list`
 - `yd-->element of the Y list`
 - `zd-->element of the Z list`
- Description:**
- Returns a coordinate from the grid at (xd, yd, zd)
- Example Usage:**
- `coord<> mins(-1,-1,-1), maxs(1,1,1);`
 - `coord<int, 3> dims(10,10,10);`
 - `Grid<UniformGrid> myG(mins, maxs, dims);`
 - `coord<> pt=myG(1,2,3);`

grids::element extraction

Function: `coord<> Point(); --> FOR ITERATORS`
`coord<> Point(double t); -->FOR ITERATORS`

Input: none

Description: returns the current point in the iteration loop for ITERATORS. Use 't' for time depedant Grids (you will need to write your own, or look to the 'RotatingGrid')

Example `Grid<UniformGrid> myG(mins, maxs, dims);`

Usage: `Grid<UniformGrid>::iterator git(myG);`
`coord<> pt=git.Point(); //will return the (0,0,0) element without looping`
`pt=git.Point(1.0); //same as above for UniformGrid`

grids::element extraction

Function:

- `double x(int i); (also r(i)) --> for Engines`
- `double y(int i); (also theta(i)) --> for Engines`
- `double z(int i); (also phi(i)) --> for Engines`

- `double x(); (also r()) --> FOR ITERATORS`
- `double y(); (also theta()) --> FOR ITERATORS`
- `double z(); (also phi()) --> FOR ITERATORS`

Input:

- nothing for Iterators
- `i--> an integer`

Description:

- returns the current point in the iteration loop along a specific direction

Example

- `Grid<UniformGrid> myG(mins, maxs, dims);`

Usage:

- `Grid<UniformGrid>::iterator git(myG);`
- `double pt=git.z(); //will return the 0th Z element without looping`
- `pt=myG.z(0);`

grids::element extraction

Function:

- `double x(double t); --> FOR ITERATORS`
- `double y(double t); --> FOR ITERATORS`
- `double z(double t); --> FOR ITERATORS`

- `double x(int i, double t); --> FOR ENGINES`
- `double y(int i, double t); --> FOR ENGINES`
- `double z(int i, double t); --> FOR ENGINES`

Input:

- `i--> the point index`
- `t--> a time`

Description:

- returns the grid value along the direction (x,y,z) assuming that the grid is TIME DEPENDANT. For the UniformGrid this function points back to `x(i)`. It is designed to allow you to write your own time dependant engine.

Example

- `coord<int> dims(5,5,5);`
- `Grid<UniformGrid> myG(mins, maxs, dims);`
- `myG.x(2, 1.0); //get the second x value at time t=1.0`
- `myG.x(2); //will be the same as above for UniformGrid`

Usage:

grids::element extraction

Function:

- `int Xdim();`
- `int Ydim();`
- `int Zdim();`

Input:

- `none`

Description:

- returns the number of steps for each direction.

Example Usage:

- `coord<int> dims(5,5,5);`
- `Grid<UniformGrid> myG(mins, maxs, dims);`
- `myG.Xdim(); //will be 5`

grids::element extraction

Function:

- double Xmin();** (also Rmin())
double Ymin(); (also Thetamin())
double Zmin(); (also Phimin())

- double Xmax();** (also Rmax())
double Ymax(); (also Thetamax())
double Zmax(); (also Phimax())

Input:

- none

Description:

- returns the smallest (min) or largest (max) value along the specific Direction (X,Y,Z) in the Grid;

Example Usage:

- `coord<> mins(-1, -2, -3), maxs(1, 2, 3);`
`Grid<UniformGrid> myG(mins, maxs, dims);`
`double pt=myG.Ymax(); //will return 2`

grids::assignments

Function: `void dim(coord<int> newDims);`

Input: `newYdim-->` the new dimension of all three dimensions

Description: `recalculates the grid components with the new dimensions`

Example Usage:

```
coord<int> dims(5,5,5);
Grid<UniformGrid> myG(mins, maxs, dims);
coord<int> newdims(10,10,10);
myG.dim(newdims); //x,y,z dims are all now 10
Grid<SurfSphereGrid> myG1(mins, maxs, dims);
myG1.dim(newdims); //changes The Y value of the dims (the master
```

grids::assignments

- Function:**
- `void Min(coord<> newMin);`
 - `void Max(coord<> newMin);`
- Input:**
- `newMin--> a new coord<>`
- Description:**
- set the new minimal or maxial values along each three directions
- Example Usage:**
- `coord<int> dims(5,5,5);`
 - `Grid<UniformGrid> myG(mins, maxs, dims);`
 - `coord<> newMax(10,10,10);`
 - `myG.Max(newMax); //The (X,Y,Z) max is now (10,10,10)`

grids::assignments

Function: `Grid operator=(Grid &rhs);`

Input: `rhs-->` A Grid WITH THE SAME ENGINE!

Description: `assigns one grid from another...THe Engine of the 'rhs' must be the same as the 'lhs'`

Example Usage: `coord<int> dims(5,5,5);
Grid<UniformGrid> myG(mins, maxs, dims);
Grid<UniformGrid> myG2=myG; //okay assignment
Grid<SurfCubeGrid> myG3=myG; //WILL FAIL (not compile)`

grids::assignments

Function:

- `void Xdim(int newdim);`
- `void Ydim(int newdim);`
- `void Zdim(int newdim);`

Input:

- `newdim--> the new value of the dimension`

Description:

- Sets the dimention along each direction.

Example Usage:

- `coord<int> dims(5,5,5);`
- `Grid<UniformGrid> myG(mins, maxs, dims);`
- `myG.Xdim(10); //The X dim is now 10`

grids::assignments

Function:

```
□ void Xmin(double newMin);  
    void Ymin(double newMin);  
    void Zmin(double newMin);  
  
    void Xmax(double newMin);  
    void Ymax(double newMin);  
    void Zmax(double newMin);
```

Input:

□ newMin--> the new minimal or maximal value

Description:

□ Sets the min or max of the grid and recalculates the entire grid.

Example Usage:

```
□ coord<int> dims(5,5,5);  
Grid<UniformGrid> myG(mins, maxs, dims);  
myG.Zmin(-5); //The Z min is now -5
```

grids::other

Function:

- `double cellVolume()`
- `double cellVolume(int i)`
- `double cellVolume(int i, double t)`

Input:

- `i`--> the point index
- `t`--> a time value

Description:

- returns ' $dx*dy*dz$ ' for a given index `i`...or for uniform grids no index is required. The 'double `t`' is used if you desire to create an engine where the grid points are time dependant. For the `Engine_t` stated here, this simply points to `CellVolume(i)`

Example

```
□ coord<int> dims(5,5,5);  
Grid<UniformGrid> myG(mins, maxs, dims);  
myG.CellVoume(); //get the voxel volume size
```

Usage:

grids::other

Function: **void Center(coord<> Cs);**
 void Center(double xC, double yC, double zC);

Input: Cs--> a coord with the values (xC, yC, zC)
xC--> The X center position
yC--> The Y center position
zC--> The Z center position=

Description: recalculates the Grid upon a recentering..simply moves the center of the grid to 'Cs'

Example recalculates the Grid upon a recentering..simply moves the center
Usage: of the grid to 'Cs'

grids::other

Function: **void Scale(coord<> mulBy);**
 void Scale(double xmul, double ymul, double zmul);

Input: mulBy--> a coord with the values (xmul, ymul, zmul)
xmove--> amount to move in X direction
ymove--> amount to move in the Y direction
zmove-->amount to move in the Z direction

Description: recalculates the Grid upon a a scaling factor..IF the the multipliers are negative then the inverse of the numbers are taken (multipliers can really only be positive)

Example `Grid<UniformGrid> myG(mins, maxs, dims);`

Usage: `myG.Scale(4,4,4); //grid is now 4 times the size it was at the start (same dims however)`
`myG.Scale(-2,-2,-2); //shrink the total grid by 1/2 (same dims however)`

grids::other

Function:

□ **int size();**

Input:

□ none

Description:

□ returns the TOTAL number of points in the grid (dimx*dimy*dimz)

Example Usage:

```
□ coord<int> dims(5,5,5);  
    Grid<UniformGrid> myG(mins, maxs, dims);  
    double s=myG.size(); //the size is 125
```

grids::other

Function: **void Translate(coord<> moveBy);**
 void Translate(double xmove, double ymove, double zmove);

Input: moveBy--> a coord with the values (xmove, ymove, zmove)
xmove--> amount to move in X direction
ymove--> amount to move in the Y direction
zmove-->amount to move in the Z direction

Description: recalculates the Grid upon a translate by some vector (xmove, ymove, zmove)

Example Grid<UniformGrid> myG(mins, maxs, dims);

Usage: myG.Translate(4,4,4); //grid has been moved by (4,4,4) in each direction

Iterators::The fast way to do loops

Here is where the speed arises, and it is also here you need to be careful about which looping method you use. There are two basic types. The iterator is meant for Speed, but it has certain restrictions...

Below is a code snipit example of how to use the iterators.

!!NOTE!!

The method above uses pointers to the Grid<> and holds a 'coord<>' already intialized so the function "Point()" is only returning the reference to the coord which is smaller in return size AND it avoids 'reinitializing' the return coord<> (which can be quite costly especially after a few million iterations)

NOTE::YOU ACSESS GRID ELEMENTS FROM THE ITERATOR!!!!

Draw backs--> You are not really sure which direction is being iterated nor the order of the points and returned...I have, in these rectangular, choosen X to be the most varing direction, and Z the slowest varying direction...There are therefor a few 'directional functions' builtin to the iterators...see the example below

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//the two corner points for a uniform rectangular grid
coord<> mins(-1, -1, -1), maxs(1,1,1);
//the number of steps between the min and max for each direction
coord<int,3> dims(10,10,10);
//a Uniform Rectangular Grid
Grid<UniformGrid> myg(mins, maxs, dims);
//sets up an iterator to loop throught the grid
Grid<UniformGrid>::iterator myIt(myg);
while(myIt){
    //the "Point()" is a "coord<double>" which prints like "x y z"
    cout<<myIt.Point()<<endl; //print the current point to the screen
    //you can use "myIt++" but it is slower then the "++myIt"
    ++myIt; //advance the iterator
}
myIt.reset(); //reset the iterator to loop again
while(myIt){
    //the "Point()" is a "coord<double>" which prints like "x y z"
    cout<<myIt.Point()<<" moo"<<endl; //print the current point to the screen
    //you can use "myIt++" but it is slower then the "++myIt"
    ++myIt; //advance the iterator
}

//-----
//reset the xdirection iterator
myIt.Xreset();
while(myIt.Xend())
```

```
{  
//prints JUST the current 'x'  
cout<<myIt.x()<<endl;  
//prints the entire Point...only the 'x' point will move here  
cout<<myIt.Point()<<endl;  
//this advances the x counter until ALL the x's have been displayed  
myIt.XadvanceGrid();  
}
```

Standard 'for' looping sequence

a simple code snippet on using the 'for' style loops

!!NOTE!!

You should AVOID using this at all possible costs (as it is slow)

```
for(int i=0;i<myg.Xdim();++i){  
    for(int j=0;j<myg.Ydim(); ++j){  
        for(int k=0; k<myg.Zdim(); ++k){  
            //again returns a coord<>  
            cout<<myg(i,j,k)<<endl;  
        }  
    }  
}
```

--Global Constants class MPIcontroller{
--- MPIworld
--Constructors
--- MPIcontroller
--- MPIcontroller
--- MPIcontroller
--- MPIcontroller
--- MPIcontroller
--- Element Extraction
--- communicator
--Other Functions
--- barrier
--- end, stop
--- get
--- getAny
--- master
--- name
--- parallel
--- put
--- rank
--- reconstruct
--- recv
--- reduce
--- scatter
--- send
--- serial
--- size
--- slave
--- splitLoop
--- start

Examples

--- get/put
--- master/slave
--- reconstruct
--- reduce

-
- This class wraps all the MPI functionality into a single class.
 - I recommend learning a bit about parallel programing before you start creating large scale programs, as you can make things hang very easily.
 - The class consists of a variety of overloaded MPI functions that work in conjunction with the data containers in the library that should be compilable with or without MPI (i.e. if you do not have MPI on one machine, but wish to compile a program that has MPI functionality, it will still compile).
 - **MPIworld** is a GLOBAL instance of MPIcontroller....you can think of it as being the "MPI_COMM_WORLD" in MPI language...or the global communicator object for all the programs
 - The MPI functionality was designed to work on 'small' parallel jobs where processor counts are of order 1-10 not 100s or 1000s (the get and put functions contain a bit too much overhead for sufficient optimization along these regards)
-

MPI::global constant

Function: □ **MPIworld**

Input: □

Description: □ This the the "Master" global controller...visible and used by the classes in the library...it needs to be started up at the the program initialization period uing the 'start' command

Example □ int main(int argv, cahr ** argc){

Usage: //start up the master
 MPIworld.start(argv, argc);
 ...do dutff...
 MPIworld.end();
 }

MPI::constructor

Function:	<ul style="list-style-type: none">□ MPIcontroller(int &argv, char *argc[]);
Input:	<ul style="list-style-type: none">□ argv-->the standard argv from the 'main' input□ argc-->the standard argc from the 'main' input
Description:	<ul style="list-style-type: none">□ starts up the master controller object from the command inputs
Example Usage:	<ul style="list-style-type: none">□ <code>MPIcontroller MyC(argv, argc)</code>

MPI::constructor

Function: `MPIcontroller(const MPI_Comm &communin);`

Input: `communin-->` an MPI communicator group

Description: creates a controller from an already initialized MPI_Comm...use this function if you have already start MPI using MPI's start up mechanism

Example `MPI_start(&argv, &argv)`

Usage: `MPIcontroller myCont(MPI_COMM_WORLD);`

MPI::constructor

Function: **MPIcontroller(const MPIcontroller &communin);**

Input: communin--> another MPIcontroller

Description: the copy constructor...creates a refrence count to the MPI_Comm being copied...

Example Usage: `MPIcontroller newC(MPIworld);`

MPI::constructor

Function: **MPIcontroller()**

Input: none

Description: the null constructor...simply initializes all the various variables to their default values...you need to start the communicator using the 'start' command after using this

Example `MPIcontroller myC;`

Usage: `myC.start(argv, argv);`

MPI::element extraction

Function: `MPI_Comm communicator();`

Input: void

Description: there is an internal data structure that is the normal MPI_Comm entity, this simply returns that structure. Only really needed if interfacing with other libraries or your own MPI code.

Example `MPI_Comm theCom=MPIworld.communicator();`

Usage:

MPI::other

Function: **int barrier();**

Input: returns any error code generated by MPI

Description: performs a barrier...meaning ALL the procs in the controller have to reach this point before the program will continue past this point....

Example MPIcontroller myMpi;

Usage: myMpi.start(argc, argv); //start an instance after declaration
myMpi.barrier(); //wait till they all get here

MPI::other

- Function:** `int end();`
 `int stop();`
- Input:** none
- Description:** end the controlers and clean up mpi
returns '1' if sucessful or it will return '0';
call this function when you no longer need MPI...typically done as the last function in a program
- Example** `MPIworld.end();`
- Usage:**

MPI::other

Function: **template<class T>**
 **int get(T &thing, int FromProc, int Tag=DEFAULT_TAG, MPI_Comm
 comm=DEFAULT_COMM);**

Input: thing--> a 'thing' an entity that can be any of the above objects
FromProc--> the processor you wish to place the Get the thing from
Tag--> a specific label you wish to give the send (Will Default if not present)
comm-->another MPI_Comm if you need one... it will use the MPIcontroller's Communicator if not specified....

Description: there are 2 basic needs when writing parallel code: get data from a proc, and put data to a proc. This is the get function. It will be able to get data to any proc 'FromProc'. This is a BLOCKING function, meaning the program will WAIT until there is a 'put' command on the 'FromProc' proc to post the data. If there is never any 'put' your program will halt here.
it will return '1' if the put was good
it will return '0' if something failed

Example MPIcontroller myMpi;

Usage: myMpi.start(argc, argv); //start an instance after declaration
int moo=6;
if(myMpi.size()>2 && myMpi.master()) myMpi.put(moo, 1);
else myMpi.get(moo, 0);

MPI::other

Function: **template<class T>**
 **int getAny(T &thing, int Tag=MPI_ANY_TAG, MPI_Comm
 comm=DEFAULT_COMM);**

Input: thing--> a 'thing' an entity that can be any of the above objects
 Tag--> a specific label you wish to give the send (Will Default if not present)
 comm-->another MPI_Comm if you need one... it will use the MPIcontroller's Communicator if not specified....

Description: a BLOCKING get (meaning if the 'FromProc' does not 'put' it, then program will hang there).
 that will wait until it gets something from ANY other processor, then return where it got it from
 it returns the processor id it Got something from

Example MPIcontroller myMpi;
Usage: myMpi.start(argc, argv); //start an instance after declaration
 int moo=6;
 if(myMpi.size()>2 && myMpi.master()){
 myMpi.put(moo, 1);
 myMpi.getAny(moo);
 }else if(myMpi.size()>2){
 myMpi.get(moo);
 if(myMpi.rank()==1) myMpi.put(moo, 0);
 }

MPI::other

Function: **bool master()**

Input: void

Description: typically, one wishes to use one proc in the controller to act as a master (the proc determining the transaction of data), this function returns whether or not the proc is the master (returns true) or not (returns false). The master is determined by a rank==0. The function is equivalent to rank()==0

Example if(MPIworld.master()) {

Usage: //send some data
 } else {
 //get the data
 }

MPI::other

Function: **std::string name()**

Input: void

Description: the name of the proc is usually not necessary unless for pretty output purposes, as the internal workings are all performed based on integer 'ranks'. This will return the name (usually the hostname) of the proc

Example std::string procName=MPIworld.name()
Usage:

MPI::other

Function: **bool parallel()**

Input: void

Description: if there is more then one proc in the controller, then this will return true (no communication nesssesary). Some alogorithms require drastically different implimentations when performed in parallel then in serial mode, this allows you to choose the proper alorithm in a transparent way.

Example if(MPIworld.parallel()) {

Usage: //do parallel algo
}

MPI::other

Function: **template<class T>**

```
int put( T &out, int To, int Tag=DEFAULT_TAG, MPI_Comm  
comm=DEFAULT_COMM)
```

Input:

- out--> the data elemtn you wish to send to the proc
- To--> the proc to send the data to
- Tag--> give the sending a unique tag to distiguish it from others...this will default to some 'DEFAULT_TAG' value if not present.
- comm--> an MPI commincator. This will default to the one inside the MPIcontroller object if not present.

Description: there are 2 basic needs when writing parallel code: get data from a proc, and put data to a proc. This

is the put function. It will be able to put data to any proc 'To'. THis is a BLOCKING function, meaning the program will WAIT until there is a 'get' command on the 'To' proc to recieve the data. If there is never any 'get' your program will halt here.
it will return '1' if the put was good
it will return '0' if something failed

Example

```
 MPIcontroller myMpi;  
myMpi.start(argc, argv); //start an instance after declaration  
int moo=6;  
if(myMpi.size()>2 && myMpi.master() ) myMpi.put(moo, 1);  
else myMpi.get(moo, 0);
```

Usage:

MPI::other

- Function:** **int rank();**
- Input:** void
- Description:** Given 4 processors, there will be 4 ranks...each proc gets a integer tag from 0..3.
 This function determines that integer and returns its value.
- Example Usage:** int myproc=MPIworld.rank()

MPI::other

Function:

- `template<class T>`
- `void reconstruct(Vector<T> &thing, int begin, int end, int masterproc=0);`
- `void reconstruct(Vector<T> &thing, Range R, int masterproc=0);`

Input:

- `thing`--> a vector of some object
- `begin`--> where the beginning of the vector was split on the specific proc
- `end`-->where the endof the vector was split on the specific proc
- `masterproc`-->where the final reconstructed vector will be

Description:

- This function attempts to piece together 'split' vectors. The other common loop splitting problem uses only sub ranges of vectors, then there is a need to reconstruct the full correct vector from the pieces distributed on the other proc...
"begin", "end", and the "R" should be the same begin,end, and Range as generated from 'splitLoop()' The correctly reconstructed vector will on the "masterproc" and you will need to use scatter() to send it back to the rest of the procs.

Example

Usage:

- see 'reconstruct example' under examples

MPI::other

Function: `template<class T>`

```
int recv(T &thing, int FromProc, int Tag=DEFAULT_TAG, MPI_Comm  
comm=DEFAULT_COMM);
```

Input:

- `thing`--> a 'thing' an entity that can be any of the above objects
- `FromProc`--> the processor you wish to place the Get the thing from
- `Tag`--> a specific label you wish to give the send (Will Default if not present)
- `comm`-->another `MPI_Comm` if you need one... it will use the `MPIcontroller`'s Communicator if not specified....

Description: there are 2 basic needs when writing parallel code: get data from a proc, and put data to a proc. This is the get function. It will be able to put data to any proc 'FromProc'. This is a NON-BLOCKING function, meaning the program will NOT-WAIT for a 'put' on the 'FromProc' and will continue after the post has been made. Use this with caution because it can create large hard-to-find problems in the code.

it will return '1' if the put was good
it will return '0' if something failed

Example

`MPIcontroller myMpi;`

`myMpi.start(argc, argv); //start an instance after declaration`

`int moo=6;`

```
if(myMpi.size()>2 && myMpi.master()) myMpi.put(moo, 1);  
else myMpi.recv(moo, 0);
```

Usage:

MPI::other

Function: **template<class T>**
 void reduce(T &thing, int ReduceType, int masterproc=0);

Input: thing--> the object you wish to reduce
 ReduceType--> 'how' to reduce the thing
 masterproc-->where the final result will be posted to. If you wish the 'thing' to be on all the procs, you must perform a 'scatter' afterwards

Description: After splitting the loop and calculating the items on different procs, one typically needs/wants to recollect the answers on each proc and use them elsewhere...this function does just that, with several basic 'Collection' operations

It should be noted that ONLY the 'masterproc' gets the correct results...thus if you want all the procs to get the total results, then use an MPIworld.scatter(thing) to put it on the rest...

ReduceType can be of these types

Reduce::Add

Reduce::Multiply

Reduce::Divide

Reduce::Subtract

Reduce::Min

Reduce::Max

Example

see the 'reduce example' under examples

Usage:

MPI::other

Function: **template<class T>**
 **int scatter(T &thing, int MasterProc=0, MPI_Comm
 comm=DEFAULT_COMM);**

Input: thing--> a 'thing' an entity that can be any of the above objects
 MasterProc--> the processor that has the 'master' data you want to give to all the other procs in the
 communicator...defaults to the 'master' proc
 comm-->another MPI_Comm if you need one... it will use the MPIcontroller's Communicator if not
 specified....

Description: a BLOCKING get/put series(meaning if the on of the procs does not get to this point, the program
 will HANG) that scatters the 'thing' to every proc in the communicator

 it returns any error code given my MPI basic put function

Example MPIcontroller myMpi;

Usage: myMpi.start(argc, argv); //start an instance after declaration
 int moo=6;
 myMpi.scatter(moo+6);

MPI::other

Function: template<class T>

```
int send(T &thing, int ToProc, int Tag=DEFAULT_TAG, MPI_Comm  
comm=DEFAULT_COMM);
```

- Input: thing--> a 'thing' an entity that can be any of the above objects
ToProc--> the processor you wish to place the 'thing' on
Tag--> a specific label you wish to give the send (Will Default if not present)
comm-->another MPI_Comm if you need one... it will use the MPIcontroller's Communicator if not specified....

- Description: there are 2 basic needs when writing parallel code: get data from a proc, and put data to a proc. This is the put function. It will be able to put data to any proc 'ToProc'. This is a NON-BLOCKING function, meaning the program will NOT-WAIT for a 'get' on the 'ToProc' and will continue after the post has been made. Use this with caution because it can create large hard-to-find problems in the code.
it will return '1' if the put was good
it will return '0' if something failed

Example MPIcontroller myMpi;

Usage: myMpi.start(argc, argv); //start an instance after declaration
int moo=6;
if(myMpi.size()>2 && myMpi.master()) myMpi.send(moo, 1);
else myMpi.recv(moo, 0);

MPI::other

Function: **bool serial()**

Input: void

Description: if there is only one proc in the controller, then this will return true (no communication nessesaray). Some alogorithms require drastically different implimentations when performed in parallel then in serial mode, this allows you to choose the proper alorithm in a transparent way.

Example if(MPIworld.serial()){
 //do serial algorithm
}else{
 //do parallel algorithm
}

Usage:

MPI::other

Function:

□ `int size();`

Input:

□ `void`

Description:

□ if there are 4 processors in your controller, the size will return '4'

Example Usage:

□ `int numporcs=MPIworld.size();`

MPI::other

Function: **bool slave();**

Input: void

Description: typically, one wishes to use one proc in the controller to cat as a master (the proc determining the transaction of data) and the others are slaves, this function returns weather or not the proc is a slave (returns true) or not (returns false). The master is determined by a rank==0. The function is equivalent to rank()!=0

Example if(MPIworld.slave()){

Usage: //get some data
 }else{
 //send the data
 }

MPI::other

Function: `Range splitLoop(int &begin, int &end, int &div, int numproc=size);`

Input:

- ❑ begin--> initially the GLOBAL begining...it will be altered to reflect the 'new' begining
- end--> initially the GLOBAL end...it will be altered to reflect the 'new' end
- div--> this will be altered to reflect the step size of the divisions.
- numproc--> will default to the controller size, but it can be an int from 0..size()

Description:

- ❑ This generates the logical indexes for a given loop on a given processor...
This can be a good starting point for quick parallelizations, but in most general conditions it is not optimal.
USE THIS ONLY for an "Equal porcessor" machine(s)...where the procs are all the same speed...if the procs are not the same speed, then you will be in an highly UNoptimized situation...use the code in the above example as a starting point for that sort of parallelization.

Example

- ❑ If we have a vector of length 10, and we have 2 procs...then
Usage: 'begin' from MPIworld.rank()==0 will be 0 and end will be 5, for
MPIworld.rank()==1 then begin=5 and end=10...div would be 5 (the
chunk size)

```
Vector<double> vect(10,0.0);
int begin=0, end=vect.size(), div=0;
Range splitR=MPIworld.splitLoop(begin, end, div);
//for MPIrank==0, begin=0, end=5, div=5, splitR=Range(0,4)
//for MPIrank==1, begin=5, end=10, div=5, splitR=Range(5,9)
```

If the (end-being) is not integer divisible by MPIworld.size(),
then the last proc gets the 'extra' pieces

```
Vector<double> vect(13,0.0);
int begin=0, end=vect.size(), div=0;
Range splitR=MPIworld.splitLoop(begin, end, div);
//for MPIrank==0, begin=0, end=6, div=6, splitR=Range(0,5)
//for MPIrank==1, begin=5, end=13, div=6, splitR=Range(6,12)
```

MPI::other

Function:

□ `int start(int &argv, char *argc[]);`

Input:

□ argv-->the standard argv from the 'main' input
argc-->the standard argc from the 'main' input

Description:

□ starts up the master controller object from the command inputs

Example Usage:

```
□ int main(int argc, char **argv) {
    //start up the MPI controller
    MPIworld.start(argv, argc);

    ...do stuff..

    //end the MPI session
    MPIworld.end()
```

simple get/put example

Shows how to use the basic MPI functions for getting and putting

!!NOTE!!

to compile, do this (assuming you have done a 'make install')::

```
blcc -mpi test.cc -o test
```

to run, do this::

```
mpirun -np 2 ./test
```

expected output would be

```
waugh.cchem.berkeley.edu rank: 0 num procs: 2
rank: 0 [ 1 4 9 16 ]
After put/get--rank: 0 [ 1 4 9 16 ]
waugh.cchem.berkeley.edu rank: 1 num procs: 2
rank: 1 [ 1 2 3 4 ]
After put/get--rank: 1 [ 1 4 9 16 ]
After scatter--rank: 0 [ 9 36 81 144 ]
After scatter--rank: 1 [ 9 36 81 144 ]
```

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv){
    MPIworld.start(argv, argc); //start up MPI subsystem

    //print out some info....
    std::cout<<MPIworld.name()<<" rank: "<<MPIworld.rank()<<" num procs: "<<MPIworld.size()<<std::endl;
    //create some vector to shuv around
    Vector<double> tmD(Spread<double>(1.0,4.0,1.0));
    Range All(Range::Start, Range::End);
    if(MPIworld.master()) tmD(All)=tmD(All)*tmD(All);
    std::cout<<"rank: "<<MPIworld.rank()<<" [ "<<tmD<<" ] "<<std::endl;
    if(MPIworld.master()){
        //put the manipulated vector to the other procs
        for(int i=1;i<MPIworld.size();++i) MPIworld.put(tmD, i);
    }else{
        //get the vector from proc 0
        MPIworld.get(tmD,0);
    }
    std::cout<<"After put/get--rank: "<<MPIworld.rank()<<" [ "<<tmD<<" ] "<<std::endl;

    // Or use the 'scatter' to distribute the new value
    if(MPIworld.master()) tmD*=9;
    MPIworld.scatter(tmD);
    std::cout<<"After scatter--rank: "<<MPIworld.rank()<<" [ "<<tmD<<" ] "<<std::endl;

    //end this very boring mpi session
    MPIworld.end();
    return 0;
}
```

This is a simple example on how to create a Master/Slave model of splitting a loop. This model is good for when you have machines of DIFFERENT or unknown speeds.

a "Master" proc simply listens for requests by any of the other procs, and send them data when they need them...it does NO calculation...so for optimum efficiency spawn one extra job then processors. If I have 2 working processor, then I'd want to do "mpirun -np 3 <myprog>" such that each proc is running full force. Doing "mpirun -np 2 <myprog>" will be the same speed (most likely slower because of communications) as NO parallelization.

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//define out function we wish to run in parallel
void MyFunction(int kk){
    cout<<endl<<"I was called on: "<<MPIworld.rank()<<" with value: "<<kk<<endl;
    sleep(MPIworld.rank()-1);
}

int main(int argc,char* argv[])
{
//Start up the Master controller
    MPIworld.start(argc, argv);

//dump out info about what and where we are
    std::cout<<MPIworld.name()<<"::"<<MPIworld.rank()<<"/"<<MPIworld.size()<<std::endl<<endl;

//this int gets sent went the Master has sent everything (the kill switch)
    int done =-1;
    int cur=0; //the current value

//if we are the master, we need to initialize some things
    if(MPIworld.master()){

//the elements in here will be sent to the slave procs
        int Max=10; //only want to send 10 things
        int CT=0, rr=-1;

//we must perform an initial send to all the proc
//from 1..size, if size>Max we need to send no more
        for(int qq=1;qq<MPIworld.size();++qq){
            MPIworld.put(CT, qq); ++CT;
            if(CT>Max) break;
        }
        int get;

//now we get an Integer from ANY processor that is NOT
// the master...and keep putting values until we run out
        while(CT<Max){
            get=MPIworld.getAny(rr); //get an int ('get'=the proc is came from)
            MPIworld.put(CT,get); //put the next value
            ++CT; //advance
        }

//put the 'We-Are-Done' flag to all the procs once we finish
        for(int qq=1;qq<MPIworld.size();++qq)    MPIworld.put(done, qq);

    }else{ //slave procs

//keep looping until we the master tells us to quit
    }
}
```

```
while(1){  
    MPIworld.get(cur,0);  
    if(cur==done) break; //id we get the kill switch get out  
    MyFunction(cur); //run out function with the gotten value  
    MPIworld.put(cur,0); //send back a request for more  
}  
}  
  
//exit MPI and leave the prog  
MPIworld.end();  
return 0;  
}
```

The reconstruct function

how to use the reconstruct function

!!NOTE!!

to compile, do this (assuming you have done a 'make install')::

```
blcc -mpi test.cc -o test
```

to run, do this::

```
mpirun -np 2 ./test
```

expected output would be

```
waugh.cchem.berkeley.edu::0/2
Original rank: 0 [ 0 1 2 3 4 5 6 7 8 9 ]
Split Parameters: Begin: 0 End: 5 data size: 10
After Addtion: rank: 0 [ 100 101 102 103 104 5 6 7 8 9 ]
waugh.cchem.berkeley.edu::1/2
Original rank: 1 [ 0 1 2 3 4 5 6 7 8 9 ]
Split Parameters: Begin: 5 End: 10 data size: 10
After Addtion: rank: 1 [ 0 1 2 3 4 205 206 207 208 209 ]
After Reconstruct: rank: 1 [ 0 1 2 3 4 205 206 207 208 209 ]
After Reconstruct: rank: 0 [ 100 101 102 103 104 205 206 207 208 209 ]
```

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc,char* argv[])
{
    MPIworld.start(argc, argv);
    std::cout<<MPIworld.name()<<":: "<<MPIworld.rank()<<"/"<<MPIworld.size()<<std::endl;

    Vector<double> CCvec(10);
    for(int i=0;i<10;++i) CCvec[i]=i;
    int b=0, e=10, div=1;

    //split the procedure
    Range toadd=MPIworld.splitLoop(b,e,div);
    cout<<"Original rank: "<<MPIworld.rank()<< " [ "<<CCvec<<" ] "<<endl;
    cout<<" Split Parameters: Begin: "<<b<<" End: "<<e<<" data size: "<<CCvec.size()<<endl;

    //add distinguishing features from each proc
    CCvec(toadd)+=(MPIworld.rank()+1)*100;
    cout<<"After Addtion: rank: "<<MPIworld.rank()<< " [ "<<CCvec<<" ] "<<endl;

    //reconstruct the data, will only be Reconstructed on the masterproc=0
    MPIworld.reconstruct(CCvec, b,e);
    cout<<"After Reconstruct: rank: "<<MPIworld.rank()<< " [ "<<CCvec<<" ] "<<endl;

    MPIworld.end();
}
```



Reduction Example--summing over a vector

!!NOTE!!

to compile, do this (assuming you have done a 'make install')::

```
blcc -mpi test.cc -o test
```

to run, do this::

```
mpirun -np 2 ./test
```

expected output would be

```
waugh.cchem.berkeley.edu rank: 0 num procs: 2
rank--0 begin: 0 end: 5 range: Range(0,4,1)
partial sum--rank: 0 15
total sum--rank: 0 30
waugh.cchem.berkeley.edu rank: 1 num procs: 2
rank--1 begin: 5 end: 10 range: Range(5,9,1)
partial sum--rank: 1 15
total sum--rank: 1 15
```

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv){
    MPIworld.start(argv, argc); //start up MPI subsystem

    //print out some info....
    std::cout<<MPIworld.name()<<" rank: "<<MPIworld.rank()<<" num procs: "<<MPIworld.size()<<std::endl;

    Vector<double> vect(10,3.0);
    int begin=0, end=vect.size(), div=0;
    //for MPIrank==0, begin=0, end=5, div=5, splitR=Range(0,4)
    //for MPIrank==1, begin=5, end=10, div=5, splitR=Range(5,9)
    Range splitR=MPIworld.splitLoop(begin, end, div);
    cout<<"rank--"<<MPIworld.rank()<<" begin: "<<begin<<" end: "<<end<<" range: "<<splitR<<endl;

    //perform the sum
    double summ=sum(vect(splitR));
    std::cout<<"partial sum--rank: "<<MPIworld.rank()<<" "<<summ<<std::endl;
    //reconstruct the master sum on the MPIrank=0
    MPIworld.reduce(summ, Reduce::Add);

    //the true sum will only be displayed from the MPIrank==0 proc
    std::cout<<"total sum--rank: "<<MPIworld.rank()<<" "<<summ<<std::endl;

    MPIworld.end();
    return 0;
}
```

Examples

--- Functional Mode

--- Interactive Mode

The CERN minimization package "MINUIT"

This is the inclusion of the entire fortran functionality of the powerful minimization program and functions 'MINUIT' from CERN. All the basic fortran functions have been included in the library so long as you have a FORTRAN 77 compiler..the best place to look for documentation about MINUIT is at CERN itself.

The initial code was taken from the source-forge project "C-MINUIT" with several major alterations and fixes to both be included in a shared library and in C++ (see source code for details about the make and header alterations)

The examples is to demonstrate to you what is necessary to run MINUIT at all. To learn about the nasty details or to learn more about its functionality, look to CERN's documentation.

!!NOTE!!

CERNS documentation:: <http://wwwinfo.cern.ch/asdoc/minuit/minmain.html>

C-MINUIT::

<http://c-minuit.sourceforge.net/>

To use MINUIT include BOTH "blochlib.h" AND "minuit/minuit.h" in your files

```
#include "blochlib.h"
#include "minuit/minuit.h"
```

There is one function YOU, as the user, MUST define before MINUIT can be used at all

```
void fcn (int npar, double* grad, double* fcval, double* xval, int iflag,
void* futil)
```

npar --> the Number of fitting parameters

grad --> the Derivative of the function with respect to the parameters (d(func)/d(param))

fcval --> this is essentially your 'chi squared' that you need to define

xval --> the place for the current value of the parameters

iflag --> various flags that function in MINUIT will set

futil --> an auxiliary function you can pass around to this function

For example, If one is fitting some data to a line, my function would be

"y=m*x + b"

npar=2

```
grad[0]=m; grad[1]=0;  
fcnval = sqrt(sum_over_i((xval[0]*x[i]+xval[1]) - y[i] )^2)  
(where 'x' is some independent variable, and 'y' is your data you wish to fit)  
iflag depends on the stage of MINUIT  
futil=NULL for this case
```

FUNCTIONAL MODE:: The second Mode, to me, is much more useful, because one can develop more complex programs, where minimization is only part of the total program...and not be a slave to the input file...however, it does require a bit of knowledge of the function available to you...which is why you should look at CERN's documentation. However, here is the same line fitting example, except using the function calls...there are numerous ways to write this program , this is just one of them.....

```
#include "blochlib.h"
#include "minuit/minuit.h"
//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

/*
//A simple example on the Functional mode of the fitting function "MINUIT"
//here we fit a line to random data (generated on the fly)
//in this directory that you will need to input pipe into stdin via
// "<progname> <query options>"

//here the '<query options>' are the options that the
//C++ part of the program asks you for upon start up
*/
//a line...
double lineFunc(double m, double b, double x)
{ return m*x+b; }

//the data vector declared GLOBAL
Vector<double> data(50);

//the master MINUIT function
void fcn (int npar, double* grad, double* fcnval, double* xval, int iflag, void* futil)
{
//NOTE: there is no iflag buisness here
// do, that is to calculate 'chi-square' or fill out 'fcnval'
// here we take x=1...50 as out independant var
    *fcnval=0;
    for(int i=0;i<data.size();++i) *fcnval+=pow(lineFunc(xval[0], xval[1], i)-
data[i], 2.0);
    *fcnval=std::sqrt(*fcnval);

}

int main(int argc, char **argv)
{
    int ss=50;
    query_parameter(argc, argv, 1, "Enter data length: ", ss);
    data.resize(ss);
//a random number generator will generate from 3 to 10
    Random<UniformRandom<> > myR(3,10);
    Range All(Range::Start, Range::End);

//initialize our data,
    data.apply(myR, All);

    int err;
    char *moo=NULL;
    MNINIT(5,6,7); //first we must initialize minuit

//get the final params
```

```

char *name1="m", *name2="b";
double startv[2]={1.0, 1.0};
double stepv[2]={1.0,1.0};
double lb=0.0, ub=0.0;

//set the two fitting parameters
// NOTE THEY START AT 1 NOT 0 (silly fortran things)
// <number> <name> <start guess> <start step size> <lower bound> <upper bound> <error flag>
MNPARM(1,name1, startv[0], stepv[0], lb, ub, err);
MNPARM(2,name2, startv[1], stepv[1], lb, ub, err);

int MAXLINE=256;
char *command; command=new char[MAXLINE];
snprintf (command, MAXLINE, "MIGRAD");

//minimize using the MIGRAD minimization type
MNCOMD (minuitfcn, command, err, NULL);

//get the final data fitted parameters
char dm[10];
MNPOUT(1, dm, startv[0], stepv[0], lb, ub, err);
MNPOUT(2, dm, startv[1], stepv[1], lb, ub, err);

//dump out the data when finished
std::ofstream oo("fitdat.m");
oo<<"vdat=[ "<<data<<" ];"<<std::endl;
oo<<"m="<<startv[0]<<" ;"<<std::endl;
oo<<"b="<<startv[1]<<" ;"<<std::endl;
oo<<"rr=1:"<<data.size()<<" ;"<<std::endl;
oo<<"fitted=rr*m+b;"<<std::endl;
oo<<"plot(rr, vdat, 'ko', rr, fitted, 'b-');"<<std::endl;
}

```

Interactive Mode

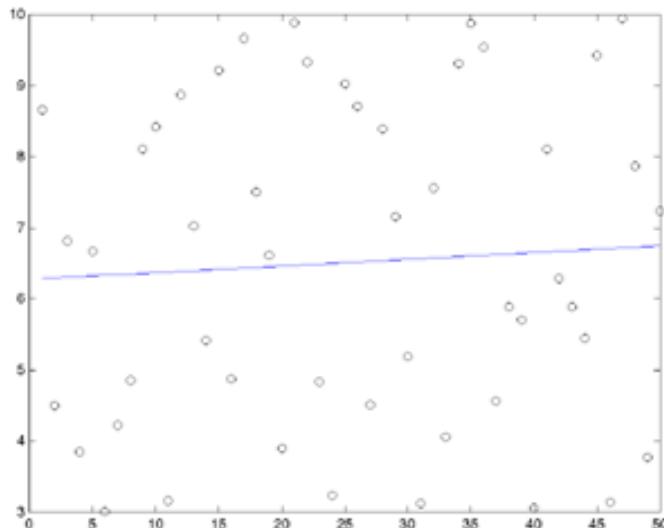
MINUIT has 2 distinct running modes...one is 'configuration-file' based (or interactive) the other is 'functional' based (specific function calls from your program).

INTERACTIVE MODE:: this mode, you define 'fcn', as above, and simply call ONE MINUIT function called 'MINUIT' in your main...this will then prompt you for a command, or you can pipe in a file into its stdin...and example is given below for fitting a line to RANDOM data in this mode...using this input set of commands

```
-----
set title
test fitting a line with random data
parameters
1 'm' 1 1 0 0
2 'b' 1 1 0 0

minimize 3000
hesse
return
```

Both the data is printed to the file, and the fitted params are printed to console (via MINUIT)



```
#include "blochlib.h"
#include "minuit/minuit.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

/*
//A simple example on the Interactive mode of the fitting function "MINUIT"
//here we fit a line to random data (generated on the fly)
//There is a 'fitting file' that MINUIT needs for the input called
//'intomin.min'
//in this directory that you will need to input pipe into stdin via
//"<progname> <query options> < intomin.min"'

//here the '<query options>' are the options that the
//C++ part of the program asks you for upon start up
```

```

*/



//a line...
double lineFunc(double m, double b, double x)
{ return m*x+b; }

//the master MINUIT function
void fcn (int npar, double* grad, double* fcnval, double* xval, int iflag, void* futil)
{
//a random number generator will generate from 3 to 10
    static Random<UniformRandom<> > myR(3,10);

//the data vector declared static to be same at every 'fcn' call
    static Vector<double> data(50);
    static Range All(Range::Start, Range::End);

//In interactive mode, the "iflag" will go between 1--4
// if ifgal==1, then it signifies the first time fcn was called,
// and so we should initialize out data
    if(iflag==1){
        data.apply(myR, All); //fill the vector with a bunch of random numbers
    }
//if the 'iflag' is not 1 or 3, then we will do what we always
// do, that is to calculate 'chi-square' or fill out 'fcnval'
// here we take x=1...50 as out independant var
    *fcnval=0;
    for(int i=0;i<data.size();++i) *fcnval+=pow(lineFunc(xval[0], xval[1], i)-data[i], 2.0);
    *fcnval=std::sqrt(*fcnval);

//if 'iflag' is 3, then it signifies the LAST call, and you shold do all
// our data outputting and clean up
    if(iflag==3){ //here we are just going to dump out to a matlab file
        std::ofstream oo("fitdat.m");
        oo<<"vdat=[ "<<data<<" ];"<<std::endl;
        oo<<"m="<<xval[0]<<" ;"<<std::endl;
        oo<<"b="<<xval[1]<<" ;"<<std::endl;
        oo<<"rr=1:"<<data.size()<<" ;"<<std::endl;
        oo<<"fitted=rr*m+b;"<<std::endl;
        oo<<"plot(rr, vdat, 'ko', rr, fitted, 'b-');"<<std::endl;
    }
}

int main(int argc, char *argv)
{
//simply call the interactive mode minuit
//minuitfcn' is the master wrapper between
// the fortran function and C++/C function 'fcn' above

//The second PArامeter would be 'futil' in the above 'fcn'
// if we needed one
    MINUIT(minuitfcn, NULL);
}

```

Examples template<class Function, class T=double, class Container=Vector<T> >

--- 1) William Rossler class bs{....

--- 2) Van-Der-Pol

--- 3) lorentz template<class Function, class T=double, class Container=Vector<T> >
class ckRK{....

template<class Function, class T=double, class Container=Vector<T> >
class stiffBS{....

- Accepted Template Params

- **Function**--> A class that need only really contain ONE thing a public function, you guessed it, call 'function' (see below about the exact format)
- **T**--> A class that is the 'type' to solve (although it best be a numerical type)...double, complex, coord, Vector, matrix (although i would not recommend a matrix...if you have a matrix already, more then likely it can be solved more efficiently via another technique (like diagonalization)), etc
- **Container**--> The most important piece...this is the list of data...it can be ANY CLASS that has these functions defined for it.."Container.resize", "max(Container)", "min(Container)", "abs(Container)", "Container + Container", "Container - Container", "Container + T", "Conatiner - T" , "Container * T", "Conatiner / T", "T+ Container", "T+Conatiner" , "T * Container", "T/Conatiner"...For most purposes this 'Conatiner' will almost always be the 'Vector' class

- This class(es) integrate large sets of arbitrariliy sized differential equations.

- There are 3 SEPARATE METHODS to perform integrations...

- **bs**--> A Bulirsch-Stoer Modified Mid Point-Richard Extrapolation method (see *Numerical Recepies in C*. Saul A. Teukolsky, *et al.*, 1997 for a nice intro to this type of method, and the nice book...Stoer, J. and Bulirsch, R. *Introduction to numerical analysis*, 1980 Springer-Verlag)....this is my personal favorite. It is fast, quite accurate, and 'almost' devoid of the need for too much fine tuning. It uses an adaptive Step size to adjust the time step to not linger around 'slow varying' times, and to take more steps during fast varying times.
- **stiffBS**--> A Bulirsch-Stoer Modified Mid PointBader-Deuflhard Semi-implicit method (see *Numerical Recepies in C*. Saul A. Teukolsky, *et al.*, 1997 for a nice intro to this type of method, and the nice book...Stoer, J. and Bulirsch, R. *Introduction to numerical analysis*, 1980 Springer-Verlag)....is the equivalent to the 'bs' above except it is really fast a good for STIFF sets of equations (equations 2 or more wildly different evolution frequencies). It requires a

- Jacobian function in the 'func' of the form "void jacobian(double time, Containter &y, rmatrix &dfdy)"
- o **ckrk** --> Cash-Karp-Runga-Kutta method (fifth order method). A standard work horse. This will probably integrate every ODE known to man with little trouble...BUT i have found it can add a some artifacts and it is fairly slow in comparison the 'bs' method...but it is much more stable, especially when it comes to 'impulses' in the diffeqs (although, no method really treats this too well without some care (i shall give you pointers later). It handles stiff ODEs okay, the BS does not at all...
-

!!NOTE!!

- This part of manual will be a bit different as there are lots of 'hidden' numerics that you, as a user, do not need to know the ins-and-outs of...instead this will be a highly example driven tutorial on how to use the integrator
 - ALL 3 CLASSES HAVE EXACTLY THE SAME FUNCTIONS AND ARE IMPLEMENTED IN EXACTLY THE SAME WAY...the only thing you need to change is the constructor string from bs mo(...) to ckrk moo(...) or to stiffbs...NOTE stiffbs requires a jacobian....
 - USE DIMENSIONLESS UNITS!!--> These methods always work best, fastest, etc when everything is on an order of magnitude of 1...you can choose not to, but if numerical precision is what you desire, dimensionless units give better results.
-

The William Rossler chaotic set of equations

You Must create a C++ class with a 'function' inside of it of the form....

void function(double t, Container &y, Container &dydt)

this function evaluates this type expression

dy/dt=f(y,t);

- **t**--> the current time step, or the independant variable
- **y**--> the current state of the system
- **dydt**--> a container that stores the results from the function evaluation.

The function class can be as simple as 4 lines, or as nasty as couple thousand lines and intertwining classes...the ODE solver just does not care so long as you give it this one function...

SEE "examples/classes/ode_example.cc" for these examples and one "WillRos"

As a simple example i will demonstrate the useage of one chaotic set of differential equations..the William-Rossler set

!!NOTE!!

The William-Rossler Attractor..... plotter over all 5000 integrated points (on my CPU (a 700 MHz pentium 3)) that integration took ~1 second. In contrast the same integration on Matlab using 'ode45' took 2.5 seconds...using 'ode15s' it took 6 second...

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//a simple class with just the 'function' and a NULL constructor
class WillRos{
public:
    WillRos(){} //Null constructor

    //the master function...The William-Rossler Attactor
    //NOTE:: eventhough 't' is not used in this function it is still nessesaray
    //for generalities sake in the 'bs' solver
    void function(double t, Vector<double> &iny, Vector<double> &dydt){
        dydt[0]=30*iny[0]-0.415*iny[0]*iny[0]-iny[0]*iny[1]-iny[0]*iny[2];
        dydt[1]=iny[0]*iny[1]-10*iny[1];
        dydt[2]=16.5*iny[2]-iny[0]*iny[2]-0.5*iny[2]*iny[2];
    }
};

//The 'main' program
int main(){
    WillRos MyDiffs; //declare a WillRoss

    //these are all the nessesaray parameters to acctually integrate something
```

```
Vector<double> IC(3);
//The initial condition...here they are all (10,5, 6)
IC[0]=10; IC[1]=5; IC[2]=6;
//this is MY sampling steps size.
//At every dt=0.1 i will write a point to a data file
double tstep=0.01;
//the starting time (0)
double startT=0;
//the ending time
double endT=50;

//declare my 'BS' solver
//(Initial Condition, The FUNCTION class)
bs<WillRos, double > odes(IC, MyDiffs);

//open an datafile
ofstream oo("data");

//create a list of times to collect points
Vector<double> times=(Spread<double>(0.0, endT, tstep));
//solve the system collecting all the data in the 'times' vector
Vector<Vector<double> > data=odes.solve(times);

//dump out all the data to a file
for(int i=0;i<data.size();i++)
    oo<<data[i][0]<<" "<<data[i][1]<<" "<<data[i][2]<<endl;
}
```

This set of equations are very stiff, meaning they have both large and small simultaneous solutions. This causes the non-stiff integrators to scream in pain because the interpolation of ODE steps become wildly variant. To solve this problem, one can use the Jacobian of the system to obtain local deviations from the current solver point. This does introduce several numerical extras that will make the integration of a non-stiff set of equations integrate much slower in comparison to the non-stiff solver. If you are unsure whether or not your equations are stiff, try integrating them with both solvers. If the stiff method takes less time, you probably have a stiff set of equations. The 'bs' solver cannot handle stiff equations at all, but the 'stiffbs' solver treats them very well. The input classes for the stiff solver require both the function AND a jacobian of the forms

void function(double t, Container &y, Container &dydt)

this function evaluates this type expression

dy/dt=f(y,t);

- **t**--> the current time step, or the independent variable
- **y**--> the current state of the system
- **dydt**--> a container that stores the results from the function evaluation.

void jacobian(double t, Container &y, matrixType &dfdy)

this function evaluates this type expression **df/dy=d(f(y,t))/dy;**

- **t**--> the current time step, or the independent variable
- **y**--> the current state of the system
- **dfdy**-->the matrix type will be determined given what your Container is...for a Vector the matrixType is a "rmatrix" for a Vector > the matrixType is ALSO 'rmatrix' with 3*the size of the vector...

!!NOTE!!

The Van Der Pol..... plotter over all 5000 integrated points (on my CPU (a 700 MHz pentium 3)) that integration took ~1.33 seconds. In contrast the same integration on Matlab using 'ode15s' it took well over 1 hour...because the algorithm used is much different (it uses a implicit backwards-difference-formula) NOTE:: using the 'bs' integrator is SUICIDE...it takes WAY TO LONG (in fact i tried to let it run for a day, but it never finished...also peaking at the data showed that the solutions were numerically unstable!)

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//a simple class with just the 'function' and a NULL constructor
class VanDerPol{
public:
    VanDerPol(){} //Null constructor

    //the master function...
    //NOTE:: eventhough 't' is not used in this function it is still nessessary
    //for generalities sake in the 'diff eq' solver
    void function(double t, Vector<double> &iny, Vector<double> &dydt){
        dydt[0]=iny[1];
        dydt[1]=1000.0*(1.0-iny[0]*iny[0])*iny[1]-iny[0];
    }

    //the master jacobian...
    //NOTE:: eventhough 't' is not used in this function it is still nessessary
    //for generalities sake in the 'diff eq' solver
    void jacobian(double t, Vector<double> &iny, rmatrix &dfdy){
```

```

dfdy.resize(2,2);
dfdy(0,0)=0.0;
dfdy(0,1)=1.0;
dfdy(1,0)=R*2.0*iny[0]*iny[1]-1.0;
dfdy(1,1)=R*(1.0-iny[0]*iny[0]);
}
};

//The 'main' program
int main(){
VanDerPol MyDiffss; //declare a WillRoss

//these are all the nessesary parameters to acctually integrate something
Vector<double> IC(2);
//The initial condition...here they are all (2,0)
IC[0]=2; IC[1]=5;
//this is MY sampling steps size. At every dt=0.1 i will write a point to a data file
double tstep=2.5;
//the starting time (0)
double startT=0;
//the ending time
double endT=10000;

//declare my 'BS' solver
//(Initial Condition, The FUNCTION class)
bs<WillRos, double > odes(IC, MyDiffss);

//open an datafile
ofstream oo("data");

//create a list of times to collect points
Vector<double> times=(Spread<double>(0.0, endT, tstep));
//solve the system collecting all the data in the 'times' vector
Vector<Vector<double> > data=odes.solve(times);

//dump out all the data to a file
for(int i=0;i<data.size();i++)
    oo<<data[i][0]<<" "<<data[i][1]<<endl;
}

```

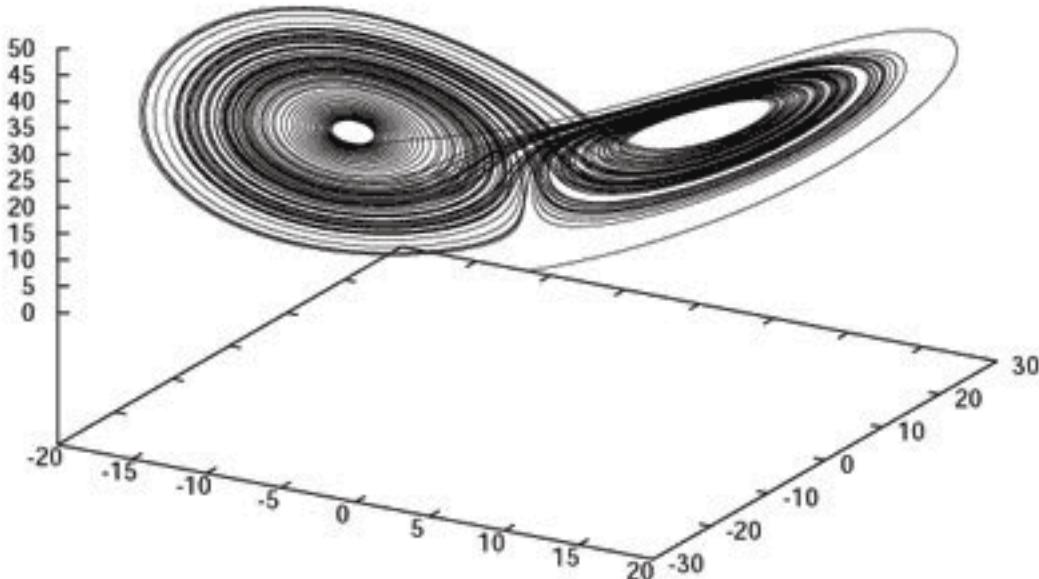
The Lorentz attractor and Lyapunov exponent

The Lorentz system is a classical chaotic system. The measure of chaos can be determined from Lyapunov exponents. These measure the deviation in trajectory if the trajectory moves 'dy' away from the current 'y.' To measure this accurately, the Jacobian is needed again, except this time not for the integration, and we need to integrate an extra set of equations...

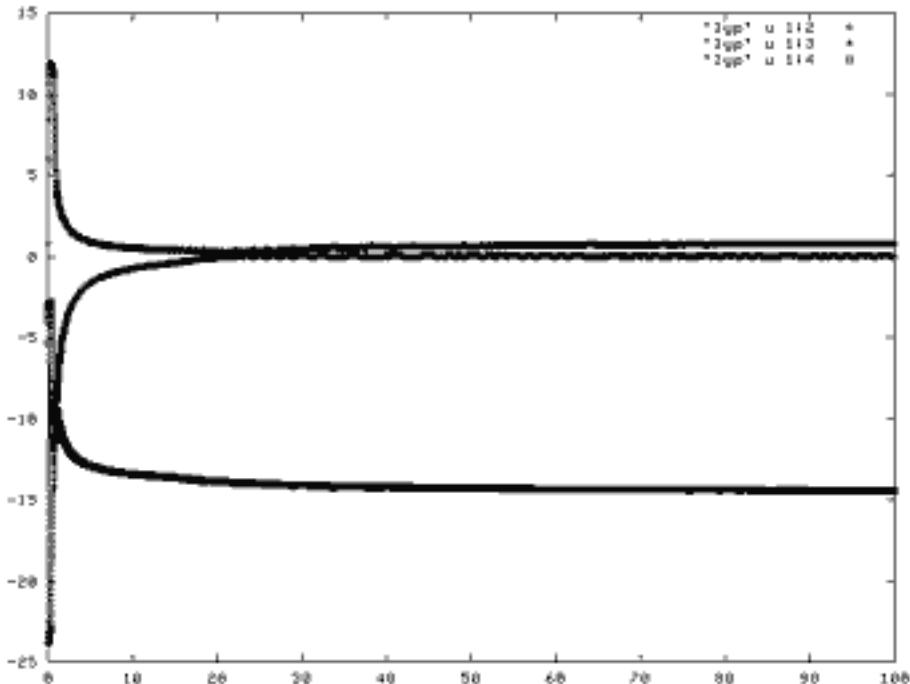
$$K'(y, t) = df(y, t)/dy * K(y, t)$$

here K is a matrix that spans each direction in the Function space...to integrate them along with the normal trajectory, we need to flatten them into the Vector $\langle .. \rangle$ format. All of these little things are taken care of by the 'Lorentz' class. This uses the "Lyapunov" class (described later)...for now just assume it is a little black box that spits out the exponents..

- This is the data generated from the below code



- The good old standard Lorenz butterfly in its yummy chaotic-ness



- Now We can a sneak peak as to the reasons for the butterfiles shape..
 - There is a LARGE negative lyapunov...i.e. One of the 3 'directions' in the set of equations shrink VERY rapidly in its size (for every time step the approximate that direction collapses to a mildly small region of space at $\sim \text{Exp}(-14)$...hits zero real quick)
 - There is a POSTIVE exponent (~ 1.5)..this means one 'direction' EXPLODES after each time step $\sim \text{Exp}(1.5)$ from its last position...this direction explores almost all of its possible phase space..
 - there is a '0' exponent...or a direction that does not change either up or down..but maintains a steady state.
- So the 'flatness' is caused by the large negative exponent...the chaotic behavior is caused by the positive one...the 'bounded' solution (i.e. the solution does not explode to infinity or DIE at zero) is caused by the 0 exponent.

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//Below is the 'function' class for the
//Lortenz Eq...note that our 'function' performs an extra
//step...it calculates the jacobian and assumes that the
//data has enough room to hold the variational equations...
class Lorentz{
public:
    rmatrix Jacobi;
    double S, R, B;
Lorentz(){
    Jacobi.resize(3,3);
}
```

```

S=10; R=28; B=8./3.;

}

void Jacobian(Vector<coord> &iny)
{
    // Lorentz jacobi
    Jacobi(0,0)=-S;
    Jacobi(0,1)=S;
    Jacobi(0,2)=0;
    Jacobi(1,0)=R-iny[0].z();
    Jacobi(1,1)=-1;
    Jacobi(1,2)=-iny[0].x();
    Jacobi(2,0)=iny[0].y();
    Jacobi(2,1)=iny[0].x();
    Jacobi(2,2)=-B;
}

void function(double t, Vector<coord> &iny, Vector<coord> &dydt){
    dydt[0].x()=S*(iny[0].y()-iny[0].x());
    dydt[0].y()=R*iny[0].x()-iny[0].y()-iny[0].x()*iny[0].z();
    dydt[0].z()=iny[0].x()*iny[0].y()-B*iny[0].z();

    Jacobian(iny);
    dydt.put(1, Jacobi*iny(1));
    dydt.put(2, Jacobi*iny(2));
    dydt.put(3, Jacobi*iny(3));
}
};

//Our Main driver now looks like.....

int main(){
    Lorentz MyDiffs;
    Vector<coord> IC(4, 1),start(4, 0);

    //set the variational bits to the
    //identity matrix to begin with
    start[1].x()=1;
    start[2].y()=1;
    start[3].z()=1;

    //time info
    double tstep=.01;
    double startT=0.;
    double endT=100.;
    double subst=tstep/10;
    int nsteps=int((endT-startT)/tstep)+1;

    bs<Lorentz, coord> odes(IC,MyDiffs);

    //output data...
    string fname="data";

```

```
ofstream oo(fname.c_str());
ofstream lyo("lyp");
Vector<coord> *out=odes.get_out();

//some initial condition
start[0].x()=10;
start[0].y()=5;
start[0].z()=6;

//A Lyapunov object (there is only 1 coord<> we care about)
Lyapunov<coord> > myLyps(1, out);

double tmS=startT;
IC=start;
odes.setInitialCondition(IC);

while(tmS<endT)
{
    odes.odeint(tmS, tmS+tstep);
    oo<<(*out)(0)<<endl;
    myLyps.CalcLyapunov(tmS, tstep);
    lyo<<myLyps;
    tmS+=tstep;
}
oo.close();
printTime();
}
```

--Constructors[--- XYZshape](#)[--- XYZshape](#)[--- XYZshape](#)[--- XYZshape](#)**--Element Extraction**[--- gridPoint](#)[--- operator\(\)](#)[--- operator\(time\)](#)[--- Point](#)[--- Point\(time\)](#)**--Assignments**[--- operator=](#)[--- setGrid](#)[--- setShape](#)**--IO**[--- printGrid](#)[--- printShape](#)**--Other Functions**[--- calculate](#)[--- empty](#)[--- gridSize](#)[--- inShape](#)[--- size](#)[--- x, y, z](#)**Examples**[--- 1\) simple rectangle](#)[--- 2\) one shape](#)[--- 3\) multi shapes](#)[--- 4\) loops](#)[--- 5\) your own shape](#)

template<class Shape_t>

class XYZshape: public Shape_t, public Grid<UniformGrid>{....

Accepted Template Params

- **Shape_t** --> another class...Like the 'Grid' class, this class uses various Engines as the basic generators for the shapes...For this class we use the UniformGrid as the main Grid generator and the 'Shape_t' as the 'cut-out' from the grid

Eventually there will be a few types of shapes...for now we have only XYZ shapes meaning they remain in the cartesian basis (as opposed to the 'spherical' basis)

This class is modeled quite closely after the Grid class, the iterator setup is the same as are most of the access functions...

- This class allows for 2 layers of grid access. As I have found from trying to make the Spherical grids 'Translate' and 'Scale' and even create an even volumetric distribution of points, it's really hard (...I know not a real excuse...), not to mention that the equations we are dealing with here in the Land of the Bloch are easily pictured and solved in the Cartesian basis (a better reason really). So here we have 'short-cut' the problem by allowing easily 'cut' out shapes from the main Grid to be utilized as the basic grid. So make cylinders, Spheres, or wacky coils are quite simple to implement and are volumetrically EVEN (in the X,Y,Z basis, and not counting edge effects)
- This class is meant for you TO ADD YOUR OWN SHAPE FUNCTIONS!! rather Create your own 'Shape_t' classes to stick in the XYZshape class...they can be as simple as One function or as complicated as you like. There is no real way for me to come up with all the shapes, so I hopefully made it easy enough for you to write and implement your own.
- With the proper function (all you need is ONE function defined in your class for the thing to work) everything from this point on should function just as I had created the Shape_t and fully integrated the sucker in the Series...even speed is not an issue as the shapes need only be generated once.
- The Basic Function you input simply takes in a point (x,y,z) that came from the UniformGrid and either Rejects it or Accepts it. (i.e. the point is either outside or inside the function)...I call this 'Validation' from here on out

Available Shapes

- **XYZfull**

- takes the Master Grid, and uses the ENTIRE thing...sort of a 'dummy' shape (no available parameter settings)

- **XYZcylinder**

- A Cylinder with the Z-axis as the main axis. You can define 'R', the 'phi' angle distribution and extend of the z axis
 - XYZcylinder(coord min, coord max) --> basic constructor
 - coord min() --> Set the minimum (R, Phi, Z) that belongs in the shape
 - coord max() --> Set the maximum (R,Phi, Z) that belong in the shape

- **XYZrect**

- Creates a rectangle inside the grid (this one is a bit silly because the UniformGrid itself is already a rectangle, but is can be used as a base class for a more complex shape)
 - XYZrect(coord min, coord max, coord center=0) --> basic constructor
 - coord min() --> Set the minimum (X,Y,Z) that belongs in the shape
 - coord max() --> Set the maximum (X,Y,Z) that belong in the shape
 - coord center() --> Set the center point (X,Y,Z) of the shape

- **XYZplaneXY**

- A Slice in the XY plane..along an arbitrary line function $y=m x + b...$
 - XYZplaneXY(double m, double b) --> basic constructor
 - double m() --> Set the slope
 - double b() --> Set the intercept
 - void SetAbove() --> set the accepted points 'Above' the plane normal
 - void SetBelow() --> set the accepted points 'Below' the plane normal

- **XYZplaneXZ**

- A Slice in the XZ plane..along an arbitrary line function $y=m x + b...$
 - XYZplaneXZ(double m, double b) --> basic constructor
 - double m() --> Set the slope
 - double b() --> Set the intercept
 - void SetAbove() --> set the accepted points 'Above' the plane normal
 - void SetBelow() --> set the accepted points 'Below' the plane normal

- **XYZplaneYZ**

- A Slice in the YZ plane..along an arbitrary line function $y=m x + b...$
 - XYZplaneYZ(double m, double b) --> basic

- constructor
 - double m() --> Set the slope
 - double b() --> Set the intercept
 - void SetAbove() --> set the accepted points 'Above' the plane normal
 - void SetBelow() --> set the accepted points 'Below' the plane normal
 - **XYZ3Dplane**
 - A Slice in the 3D plane..along an arbitrary line function
 $(x,y,z)=(mx,my,mz)*(x,y,z)+(bx,by,bz)\dots$
 - This one is hard to visualize in ones head but it allows for a generic plane slice
 - XYZplaneYZ(coord m, coord b) --> basic constructor
 - coord m() --> Set the slope
 - coord b() --> Set the intercept
 - void SetAbove() --> set the accepted points 'Above' the plane normal
 - void SetBelow() --> set the accepted points 'Below' the plane normal
-

!!NOTE!!

PLEASE SEE THE EXAMPLES

There are 2 basic Ways of Generating complex grid shape patterns. The first is to use the 'XYZblaa' functions above, which is the simplest. However more complex shapes are sometimes desired and are, in fact, combos of the 'XYZblaa' functions above (i.e. a rectangle and a cylinder in one grid array). So i have made a few operators and functions that use the '&&&' and '||' (AND and OR) operators to allow you to produce these combo shapes.

NOTE:: THE DOWN SIDE!!-->These expression of multi-shapes CANNOT be declared as a new variable (in order for me to do this i would had to make you make all your shape decalarations global and that is a nasty thing to have to do)...what that means is that you must type out the expression every time you need to use it...as an example...

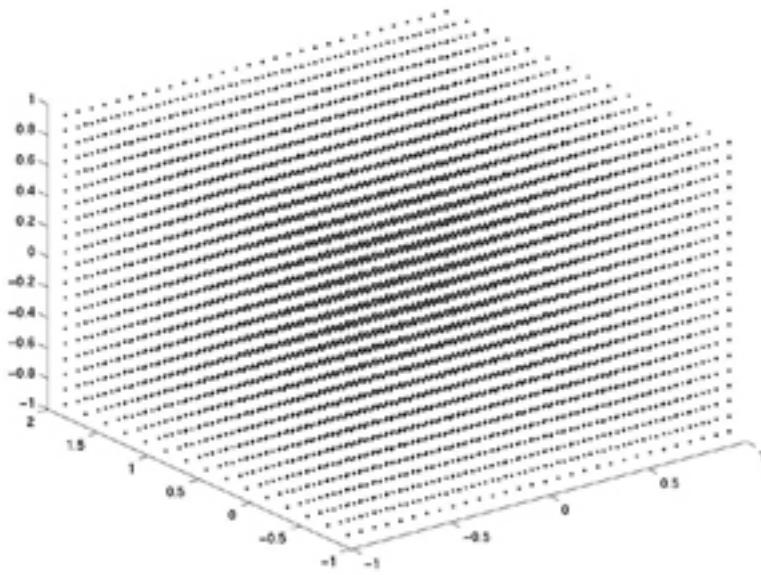
MyShape=XYZrect || XYZcylinder; //This assignment CANNOT BE DONE and will NOT compile

void {your func name}(XYZrect || XYZcylinder); //this will work fine

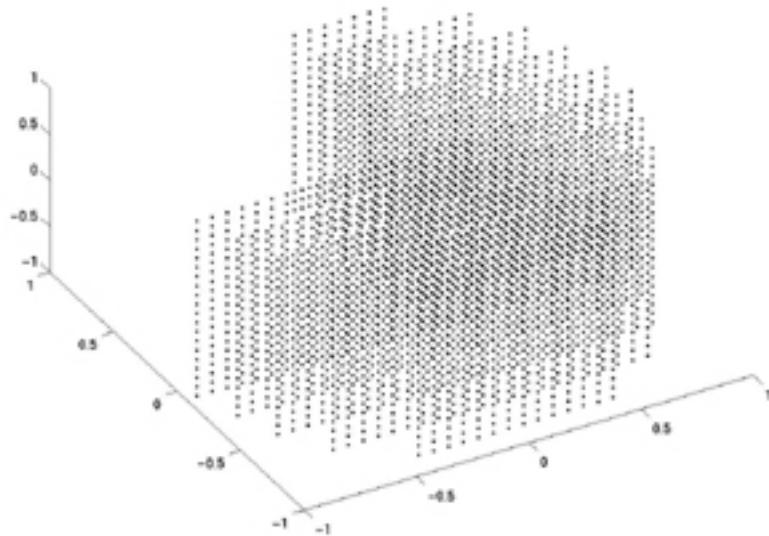
- **Examples of the 'Normal/Singluar' Shapes..(in the form of 'code snippit' and 'picture')**

- ```
//set up the basic uniform grid layout..
coord<> mins(-1,-1,-1), maxs(1,2,1);
coord<int> dims(30,15,20);
Grid<UniformGrid> BasicG(mins,maxs,dims);
```

```
//set up the XYZshape functions
//default 'full' grid
XYZshape<> MasterGrid(BasicG, XYZfull());
```



```
XYZcylinder mys(0,1.0, 0, 3.0*PI/2.0, -2,2);
XYZshape<XYZcylinder> MasterGrid(BasicG,mys);
```



- Examples of the 'MULTI' Shapes..(in the form of 'code snippet' and 'picture')

- The general syntax for these must be followed..

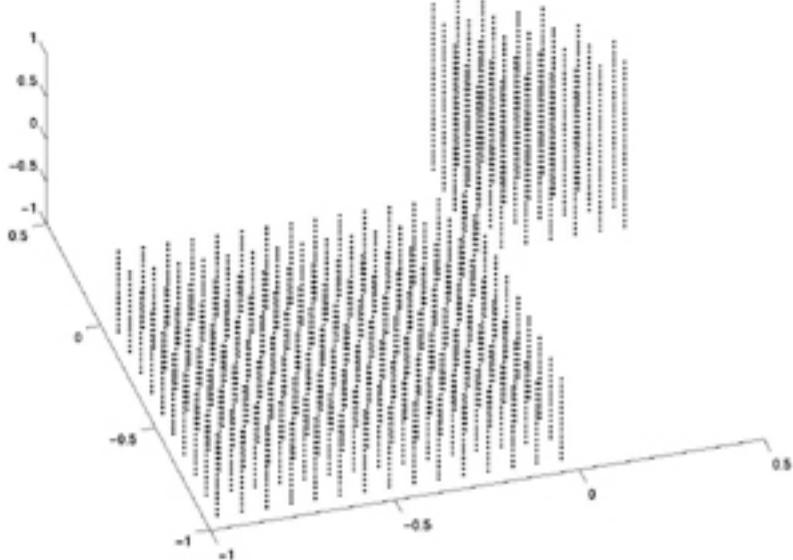
```
//set up the basic uniform grid layout..
coord<> mins(-1,-1,-1), maxs(1,2,1);
coord<int> dims(30,15,20);
Grid<UniformGrid> BasicG(mins,maxs,dims);
```

```

//set up the XYZshape functions
XYZrect rect(coord<>(-1,-3,-1),
coord<>(0,0,0)); //rectanlg shape
XYZplaneXY plane(5, 1); //an XY plane
plane.SetBelow();
XYZcylinder cyl(0,0.5, 0, PI/2, -2, 2); //a
cylinder
XYZshape<> MasterGrid(BasicG); //NOTE:: no
'XYZshape' placed in here...

//THIS IS HOW TO CALCULATE THE GRIDS....
//the means..."thing in both the rect OR in
the cyl"
MasterGrid.calculate(rect || cyl);

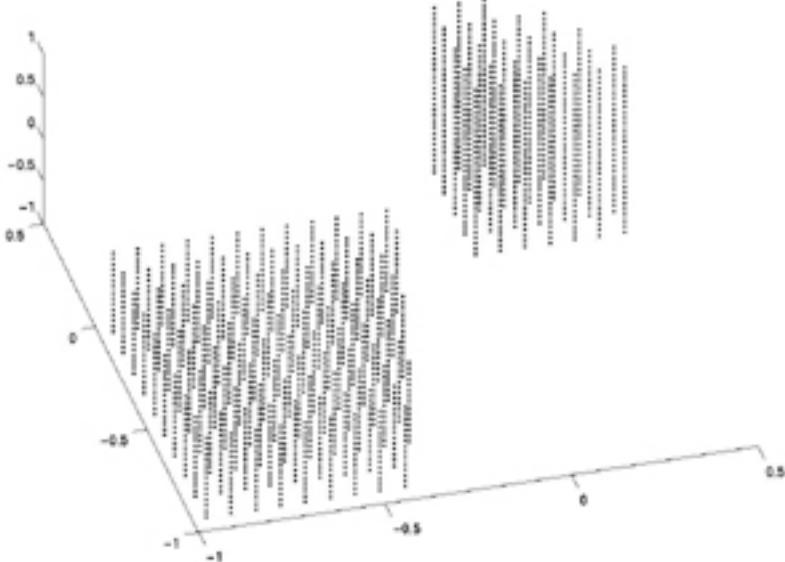
```



```

//the means..."thing in both the rect AND
plane OR in the cyl"
MasterGrid.calculate(plane && rect || cyl);

```



---

## Shapes::constructor

**Function:**  **XYZshape()**

**Input:**  none

**Description:**  empty constructor

**Example Usage:**  XYZshape<> moo; //this does nothing

## Shapes::constructor

**Function:**  **XYZshape**(**const XYZshape**  $\circ$ )

**Input:**  cp--> another XYZshape of the same shape\_t

**Description:**  copies one shape WITH THE SAME 'Shape\_t'

**Example Usage:**  **XYZshape<XYZfull>** sg();  
**XYZshape<XYZfull>** sg2(sg); //okay to copy  
**XYZshape<XYZrect>** ug1(sg); //WILL FAIL incompatible engines

## Shapes::constructor

**Function:** `XYZshape(const Shape_t &MyShape, const Grid<UniformGrid> &MyGrid)`

**Input:** `MyShape-->A shape class (a XYXfull, XYZrect, etc)`  
`MyGrid--> A uniform grid`

**Description:** `This the master constructor`  
`upon intialization of this constructor the Shape Grid will acctually be calculated`

**Example** `coord<> mins(0, 0, 2.1), maxs(1, PI2, 3.0);`

**Usage:** `XYZcylinder myS(mins, maxs); //make the shape`

`coord<> gmins(-1,-1,-1); gmaxs(4,4,4);`

`coord<int> dims(20,20,20);`

`Grid<UniformGrid> ugl(gmins, gmaxs, dims); //make the uniform grid`

`XYZshape<XYZcylinder> MyShape(myS, ugl); //The entire grid and`  
`shape are now made`

## Shapes::constructor

**Function:** `XYZshape(const Grid<UniformGrid> &MyGrid)`

**Input:** `MyGrid-->` A uniform grid

**Description:** `sets the master uniform grid to the input, and calculates the shape assuming a default 'Shape_t'`

**Example** `coord<> gmins(-1,-1,-1); gmaxs(4,4,4);`

**Usage:** `coord<int> dims(20,20,20);`

```
Grid<UniformGrid> ugl(gmins, gmaxs, dims); //make the uniform grid
XYZshape<> MyShape(ugl); //The entire grid and shape
```

## Shapes::element extraction

**Function:** `coord<> gridPoint(int xpos, int ypos, int zpos);`

**Input:** `xpos, ypos, zpos-->` the position in the Grid list

**Description:** `extracts the Grid Point from the 'Grid<UniformGrid>' under layer...NOT related to the Shape grid points`

**Example** `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:** `coord<> pt=myG.GridPoint(1,1,1); //extracts the 1,1,1 point from the grid`

## Shapes::element extraction

**Function:**  `coord<> operator(int pos);`  
 `coord<> operator(int xd, int ys, int zd);`

**Input:**  pos--> the current list position  
xd, yd, zd--> the position of each element desired

**Description:**  returns the coord<> at the position 'pos' or at (xd, yd, zd). Using (xd, yd, zd) is quite slow after many iterations.

**Example**  `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:**  `coord<> pt=myG(1,2,3);`  
`pt=myG(1); //the second shape point extracted`

## Shapes::element extraction

**Function:**  `coord<> &operator()(int i, double t)`  
`inline coord<> operator()(int i, int j, int k, double t)`

**Input:**  i,j,k--> the shape element index  
t--> the time

**Description:**  If either the shape or the grid is time dependant, this will return the point at index 'i' at time 't.'  
Currently there are no Time depdant shapes except fro 'RotatingGrid.'  
if there is not time dependance it will return 'operator(i)'

**Example**  `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:**  `coord<> pt=myG(1,5.0); //a point a t=5.0`  
`pt=myG(1); //the same as above`

## Shapes::element extraction

**Function:** `coord<> Point(); --> FOR ITERATORS`

Input: `none`

Description: `returns the current point in the iteration loop`

Example `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

Usage: `XYZshape<XYZcylinder>::iterator git(myG);  
coord<> pt=git.Point(); //will return the (0,0,0) element without  
looping`

## Shapes::element extraction

**Function:**

- `coord<> Point(double time); --> FOR ITERATORS`
- `coord<> Point(int i, double t); --> FOR SHAPE`
- `coord<> Point(int i, int j, int k, double t) --> FOR SHAPE`

**Input:**

- i,j,k--> the shape element index
- t--> the time

**Description:**

- If either the shape or the grid is time dependant, this will return the point at index 'i' (or i,j,k) at time 't.' Currently there are no Time depdant shapes except fro 'RotatingGrid.'
- if there is not time dependance it will return 'operator(i)'

**Example**

```
XYZshape<XYZcylinder> myG(MYshape, MYgrid);
coord<> pt=myG.Point(1,5.0); //a point at t=5.0
```

**Usage:**

```
pt=myG.Point(1); //the same as above
```

## Shapes::assignments

- Function:** `XYZshape operator=(XYZshape &rhs);`
- Input:** `rhs-->` A Shape WITH THE SAME Shape ENGINE!
- Description:** `assigns one shape from another...THe ShapeEng of the 'rhs' must be the same as the 'lhs'`
- Example Usage:** `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`  
`XYZshape<XYZcylinder> myG2=myG; //okay assignment`  
`XYZshape<XYZrect> myG3=myG; //WILL FAIL (not compile)`

## Shapes::assignments

**Function:** `void setGrid(Grid<UniformGrid> &rhs);`

**Input:** `rhs-->` A Uniform Grid

**Description:** `this will assign the Uniform grid type (if you wish to change it)`  
`it will "recalculate" the list inside 'XYZshape'`

**Example** `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:** `myG.setGrid(MyGrid2); //sets a new uniform grid AND recalcs the shape points`

## Shapes::assignments

**Function:** `void setShape(Shape_t &rhs);`

**Input:** `rhs-->` A Shape Engine...must be the same as the XYZshapes Shape type

**Description:** `this will assign a new Shape Engine`  
`it will "recalulate" the list inside 'XYZshape'`

**Example** `XYZcylinder MYshape(0,1,0,PI2, 0, 1), Myshape2(.5,1, 0, PI, 0,1);`

**Usage:** `XYZrect Myrect(-1,1,-1,1,-1,1);`

```
XYZshape<XYZcylinder> myG(MYshape, MYgrid);
myG.setShape(Myshape2); //sets a new shape engine AND recalcs the
shape points
myG.setShape(Myrect); //This will FAIL
```

## Shapes::IO

**Function:** `void printGrid(ostream &out);`

**Input:** `out-->` an output stream (a file, a console, etc)

**Description:** `prints the points valid to the ENTIRE GRID to "out" in the 3 column format ASCII  
<x point> <y point> <z point>`

**Example** `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:** `ofstream oufile("lookhere");  
myG.printGrid(oufile); //will write the grid to the file "lookhere"`

## Shapes::IO

**Function:** `void printShape(ostream &out);`

**Input:** `out-->` an output stream (a file, a console, etc)

**Description:** `prints the points valid to the shape to "out" in the 3 column format ASCII  
<x point> <y point> <z point>`

**Example** `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:** `ofstream oufile("lookhere");  
myG.printShape(oufile); //will write the shape to the file  
"lookhere"`

## Shapes::other

**Function:** `void calculate(const ShapeExpr &in);`

**Input:** `in` --> a Shape\_t or the multi-combinations of shapes

**Description:** calculates the Points in the shape based on the 'ShapeFunc' inside the 'ShapeExpr'

**Example** `XYZcylinder MYshape(0,1,0,PI2, 0, 1), Myshape2(.5,1, 0, PI, 0,1);`

**Usage:** `XYZrect Myrect(-1,1,-1,1,-1,1);`

`XYZshape<Myrect> myG(MYgrid);`

`myG.calculate(Myrect);`

## Shapes::other

**Function:**  **bool empty()**

Input:  none

Description:  If either 1) the shape has not been calculated or 2) there are no points in the Shape this will be true

Example  XYZshape<XYZcylinder> myG;

Usage:  myG.empty(); //will be true

## Shapes::other

**Function:**  `int gridSize();`

**Input:**  none

**Description:**  returns the size of the underlying 'Grid<UniformGrid>' not the shape

**Example Usage:**  `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`  
`int sizes=myG.gridSize(); //will return the total size of MYgrid`

## Shapes::other

**Function:**  `bool inShape(coord<> testpt);`  
`bool inShape(double x, double y, double z);`

**Input:**  `testpt-->`a point to test weather or not the ShapeEngine will accept this point  
`x-->` the x position  
`y-->` the y position  
`z-->` the z position

**Description:**  Is simply a linker to the Shape frunctions "ShapeFunc" it will return TRUE if the point is within the shape  
it will return FALSE if the point in NOT within the shape

**Example**  `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:** `bool sizes=myG.inShape(3,4,5); //is the point (3,4,5) in the shape?`

## Shapes::other

**Function:**  **int size();**

**Input:**  none

**Description:**  returns the size of the shape (the number of points that belong inside the shape)

**Example Usage:**  XYZshape<XYZcylinder> myG(MYshape, MYgrid);  
int sizes=myG.size(); //will return the total size

## Shapes::other

**Function:**

- `double x(int i); --> FOR SHAPE`
- `double y(int i); --> FOR SHAPE`
- `double z(int i); --> FOR SHAPE`
  
- `double x(); --> FOR ITERATORS`
- `double y(); --> FOR ITERATORS`
- `double z(); --> FOR ITERATORS`

**Input:**

- `i-->` The i'th element in the z list

**Description:**

- for SHAPES
  - returns the the (X,Y,Z) value at the shape point 'i'
- for ITERATORS
  - returns the current Z point in the iteration loop

**Example**

- `XYZshape<XYZcylinder> myG(MYshape, MYgrid);`

**Usage:**

- `XYZshape<XYZcylinder>::iterator git(myG);`
- `coord<> pt=git.z(); //will return the (0,0,0) element without looping`
- `double pt=myG.x(3); //will return the 3rd x element`

## First Examples of the 'Normal/Singular' Shapes.

---

This generates a simple rectangular grid using the 'Grid' class as the master generator

---

```
//set up the basic uniform grid layout..
coord<> mins(-1,-1,-1), maxs(1,2,1);
coord<int> dims(30,15,20);
Grid<UniformGrid> BasicG(mins,maxs,dims);

//set up the XYZshape functions
//default 'full' grid
XYZshape<> MasterGrid(BasicG, XYZfull());
```

---

## Using one XYZ shape function

---

This uses the 'XYZcylinder' to provide a rectangular grid int the shape of a cylinder (or a 'pie')

---

```
//set up the basic uniform grid layout..
coord<> mins(-1,-1,-1), maxs(1,2,1);
coord<int> dims(30,15,20);
Grid<UniformGrid> BasicG(mins,maxs,dims);

//set up the XYZshape functions
//using a cut away cylinder
XYZcylinder mys(0,1.0, 0, 3.0*PI/2.0, -2,2);
XYZshape<XYZcylinder> MasterGrid(BasicG,mys);
```

---

## Using the || and && operators to use combine multiple shapes

---

The general syntax for these must be followed..

---

```
//set up the basic uniform grid layout..
coord<> mins(-1,-1,-1), maxs(1,2,1);
coord<int> dims(30,15,20);
Grid<UniformGrid> BasicG(mins,maxs,dims);

//set up the XYZshape functions
XYZrect rect(coord<>(-1,-3,-1), coord<>(0,0,0)); //rectanlg shape
XYZplaneXY plane(5, 1); //an XY plane
plane.SetBelow();
XYZcylinder cyl(0,0.5, 0, PI/2, -2, 2); //a cylinder
XYZshape<> MasterGrid(BasicG); //NOTE:: no 'XYZshape' placed in here...

//THIS IS HOW TO CALCULATE THE GRIDS....
//the means..."thing in both the rect OR in the cyl"
MasterGrid.calculate(rect || cyl);

//the means..."thing in both the rect AND plane OR in the cyl"
MasterGrid.calculate(plane && rect || cyl);
```

---

## Using the iterators for Looping through the grid

Here is where the speed arises, and it is also here you need to be careful about which looping method you use. There are two basic types. One is meant for hyper Speed, but it has certain restrictions...the other is the more basic but is slower...

- There are TWO DIFFERENT ITERATORS...
  - 1)The Shape Iterator:: XYZshape::iterator this loops over ONLY those grid points that were 'validated' by your Shape\_t It behaves exactly like the iterator from the Grid class
  - 2) The Entire Grid Iterator:: XYZshape::Griditerator This is just a renaming of the Grid::iterator This Behaves exactly like the iterator from the Grid class (becuase it IS exactly the same thing)
- Below are some examples....(see the 'Grid' class for using the "Griditerator", although you will notice there is almost no difference then this except the inital nameing)
- NOTE:: Dues to the nature of the 'Shape' Validation (and becuase shapes ARE NOT generaly uniform in any one direction there is NO DIRECTION ITERATORS for the Shape! Not even for 'method 2.' there is only One way...that is Down the list

```
//the two corner points for a uniform rectangular grid
coord<> mins(-1, -1, -1), maxs(1,1,1);

//the number of steps between the min and max for each direction
coord<int,3> dims(10,10,10);

//a Uniform Rectangular Grid
Grid<UnifromGrid> myg(mins, maxs, dims);

//a Shape_t class that generates a cylinder
XYZcylinder myCyl(0,1,0,P2/, 0,1);

//i have now created a Shape and a Grid bilayer
//...all set up a ready to go
XYZshape<XYZcylinder> masterGrid(myCyl, myg);

//sets up an iterator to loop throught the SHAPE (NOT the UniformGrid Grid)
XYZshape<XYZcylinder>::iterator myIt(masterGrid);

//this will output just the Validated shape points
while(myIt){
 //the "Point()" is a "coord<double>" which prints like "x y z"
 //print the current point to the screen
 cout<<myIt.Point()<<endl;
 //you can use "myIt++" but it is slower then the "++myIt"
 //++myIt; //advance the iterator
}
//reset the iterator to loop again
myIt.reset();
while(myIt){
 //the "Point()" is a "coord<double>" which prints like "x y z"
 //print the current point to the screen
 cout<<myIt.Point()<<" moo"<<endl;
 //you can use "myIt++" but it is slower then the "++myIt"
 //++myIt; //advance the iterator
}

//now we will use the 'Griditerator'...which will output the entire "UniformGrid"
XYZshape<XYZcylinder>::Griditerator myGIT(masterGrid);
while(myGIT){
```

```
//the "Point()" is a "coord<double>" which prints like "x y z"
//print the current point to the screen
 cout<<myGIIt.Point()<<endl;
//you can use "myIt++" but it is slower then the "++myIt"
 ++myGIIt; //advance the iterator
}

//Using the direction iterators of the 'Shape' iterator CANNOT HAPPEN
//the code below WILL NOT COMPILE!
//However the Directional iterations from the Griditerator still work as before!
/*
myIt.Xreset(); //reset the xdirection iterator
while(myIt.Xend())
{
 cout<<myIt.x()<<endl;
 cout<<myIt.Point()<<endl;
 myIt.XadvanceGrid();
}
*/
//remember there is NO directional loops for the shapes!
//This is the normal 'for' looping sequence
for(int i=0;i<masterGrid.size();++i){
 cout<<myg(i)<<endl; //again returns a coord<>
}
```

---

## How to write your own Shape Generation Engine

The easiest way to show you how to write a shape is to show you how simple the built in types are i will show you "XYZcylinder"....The comments below will contain most of the 'things you must do' to get the shape working

```
//you need to Include this file...it will be in the path "{source-top}/src/container/grids/"
#include "xyzgenshape.h"

//start with a class definintion...and make public the "GeneralXYZShape" class
//as shown below..This acts a the container for the 'Min' 'Max' and 'Center' coord's
class XYZcylinder_ : public GeneralXYZShape {

 public:
 //null constructor..ALWAYS good practice to put this here even if it does NOTHING
 XYZcylinder_(){}

 //copy constructor..ALWAYS good practice to put this here even if it does NOTHING
 XYZcylinder_(XYZcylinder_ & copy):
 min_(copy.min_), max_(copy.max_), center_(copy.center_)
 {}

 //the basic constructor for this class takes in the Min (r,phi, z) and the Max(r,phi,z)
 XYZcylinder_(coord<> &minin, coord<> &maxin):
 min_(minin), max_(maxin), center_(0)
 {}

 //the basic constructor for this class takes in the Min (r,phi, z) and the Max(r,phi,z)
 //in the non-coord style
 XYZcylinder_(double rmin, double rmax, double phimin, double phimax, double zmin, double zmax):
 min_(rmin, phimin, zmin), max_(rmax, phimax, zmax), center_(0)
 {}

 //THESE ARE THE TWO FUNCTIONS YOU HAVE TO HAVE
 //THESE ARE THE TWO FUNCTIONS YOU HAVE TO HAVE
 //i repeat....
 //THESE ARE THE TWO FUNCTIONS YOU HAVE TO HAVE
 //***You Must CALL THEM "ShapeFunc"***
 //----They take in an XYZ point and return A TRUE or FALSE
 //----TRUE--> if that point belongs inside the shape
 //----FALSE--> if that point does NOT belong inside the shape

 //this one takes in a grid point as the 'x, y, z'
 bool ShapeFunc(double x, double y, double z)
 {
 //this is the 'r' value of the input point
 double tmr=sqrt(x*x+y*y);

 //this is the phi angle of the input point
 double tmph=0.0;
 if(tmr!=0) tmph=acos(y/tmr);
 if(x<0.0){ tmph=PI2-tmph; }

 //remeber the user was to input "r, Phi, z" into the constructors
 //although here it looks like i am comparing "x" to "r" remeber the extraction element
 //the first entry in a coord<> is "x()" and so on
 if(tmr>=min_.x() && tmr<=max_.x())
 {
 if(tmph>=min_.y() && tmph<=max_.y())
 {
 if(z>=min_.z() && z<=max_.z())
 {
 return true; //okay, this point is within the limits...
 }
 }
 }
 }
}
```

```
 return false; //this point failed the test...
}
//this one takes a grid point as the coord(x,y,z)'
bool ShapeFunc(coord<> &xyz)
{
//here all we are doing is overloading the ShapeFunc and diverting it to the one above
 return ShapeFunc(xyz(0), xyz(1), xyz(2));
}
};

//Now you Must 'Register' the shape using this typedef
// this allows the expression like 'Shape1 || Shape2'
// to function properly
typedef XYZparts<XYZcylinder_> XYZcylinder;
```

---

**--Constructors****--- TimeTrain****--- TimeTrain****--- TimeTrain****--Element Extraction****--- beginStepTime****--- beginTime****--- dt****--- endStepTime****--- endTime****--- step****--- stepsize****--- t****--Assignments****--- addStep****--- dropStep****--- operator=****--- operator=****--- setBeginTime****--- setEndTime****--- SetEngine****--- setStep****--- setSubStep****--IO****--- display****--- operator<<****--- operator<<****--- operator<<****--- operator>>****--- operator>>****--- print****--- printFileTimEngine****--- read****--- read****--- read, readASCII****--- write****--- write, writeACSI****--Other Functions****--- size****Examples****--- 1) looping**

```
template<class Engine_t>
class TimeTrain: public Engine_t{....
```

**Accepted Template Params**

- **Engine\_t** --> another class...The distribution engine. This class acts as the generator type for the time Train

- You can think of this class a a specialized "Range" container simple stores A Begining-- and End with the appropriate stepsizes
- As with "Shape" and "Grid" this too requires a generator engine that generates the train of points
- This is acutually just a nice extension to the Vector class that contains a few more bits of information that are critical to the ODE solvers.
- It simply generates a 2 column list {time} {step divisions}
- Each {time} is separated by the Engines criterion, the {step divisions} are for the ODE solver...this is the initial guess as to the number of divisions to chop up {time1}...{time2}. The ODE solver has adaptive step sizes, BUT if the {step divisions} is too large you will get BAD results from the ODEsolver..

Builtin Engines...(things that i have written already)

**• ConstUniformTimeEngine**

Use this when you know the exact number of divisions you wish to chop up the time sequence into. It is faster then the UniforTimeEngine becuase loop unrolling can occur, but it is a specialized Engine (i.e. ou cannot change 'N'). Also all the divisions between time steps ARE THE SAME (hence the 'Uniform')

**CONSTRUCTOR-->(Start Time, End Time, Num Sub Step Size)**

```
//A 128 point time train starting at
0..to 0.2 with a substep size of 100
ConstUniformTimeTrain<128> ct(0, 0.02,
100);
```

**• UniformTimeEngine**

Like the Const version above accept that you can change and modify its length, divisions, etc at any time.

**CONSTRUCTOR-->(Start Time, End Time, NumSteps, Num Sub Step Size)**

```
//A 100 point time train starting at
0..to 0.2 with a substep size of 128
```

```
UniformTimeTrain ct(0, 0.02, 100, 128);
```

- **FileTimeEngine**

Reads in a Two column file {time} {substeps} it accepts comment lines either starting with '#' or the matlab comments '%' (so you can read the same file into matlab without any troubles)

### **CONSTRUCTOR-->(File Name)**

```
//reads in the file "mytimes"
FileTimeEngine myT("mytimes");
```

## **Writing Your Own Engine**

- Building engines are not as easy as building the Shape Engines..becuase there can be a varaity of supplimental functions that you could write...in fact all these 'Assignement' functions and all these 'IO' functions need to be written for that class (or not written just do not expect to use them in your code)
  - Function you could/should write for Engines
    - Assignments
      - setStep(), setSubStep(), setBeingTime(),  
setEndTime(), addStep(), dropStep()
      - Any conversion operators, operator=( ) (one of the  
Most impotent ones in the class)
    - IO function
      - read, readASCII, write, writeASCII, print
  - Look to the "unitrain.h" and "consttrain.h" files for the methods and set up....
-

## TimeTrain::constructor

**Function:**  **TimeTrain()**

**Input:**  none

**Description:**  empty constructor

**Example Usage:**  `TimeTrain<UniformTimeTrain> moo( ); //this does nothing`

## TimeTrain::constructor

**Function:**  **TimeTrain(const TimeTrain &cp)**

**Input:**  cp--> another TimeTrain of the same type

**Description:**  the copy constructor

**Example**  UniformTimeTrain Eng(0, 1, 10,100);

**Usage:**  TimeTrain<UniformTimeTrain> sg(Eng); //A valid Time train

TimeTrain<UniformTimeTrain> sg2(sg); //okay to copy

TimeTrain<ConstUniformTimeTrain> ug1(sg); //WILL FAIL incompatible engines

## TimeTrain::constructor

**Function:** `TimeTrain(const Engine_t &Eng)`

**Input:** `Eng`--> The Time Train Engine

**Description:** `This is the master constructor.`

Upon initialization of this constructor the Time Train will actually be calculated.

**Example** `//a uniform train starting at t=0, ends at t=1, in 10 step`

`divisions`

`// with a sub division of 100`

`UniformTimeTrain Eng(0, 1, 10,100);`

`TimeTrain<UniformTimeTrain> sg(Eng); //A valid Time train`

`//or alternatively..in one line`

`TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`

**Usage:**

## TimeTrain::element extraction

**Function:**  `double beginStepTime(int i);`  
`double beginStepTime(); --> FOR ITERATORS`

**Input:**  `i-->`the index from which to grab a time

**Description:**  Returns the the being time step at the curent iterator position or at index 'i.'

**Example**  `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`  
**Usage:**  `double t1=sg.beginStepTime(4); //grabs the fifth time`  
`TimeTrain<UniformTimeTrain>::iterator myIt(sg);`  
`t1=myIt.beginStepTime(); //grabs the First element (here it would`  
`be 0) as we have not iterated yet`  
`++myIt;`  
`t1=myIt.beginStepTime(); //we are now at he second element in the`  
`list (which is 0.1)`

## TimeTrain::element extraction

**Function:**  **double beginTime()**

**Input:**  void

**Description:**  Returns the begining time

**Example**  TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));

**Usage:**  double t1=sg.beginTime(); //grabs the begining time (here it is 0)

## TimeTrain::element extraction

**Function:**  **double dt(int i);**  
**double dt(); --> FOR ITERATORS**

**Input:**  i-->the index from which to grab a time

**Description:**  Returns the the delta T at index 'i.'  
This is "beginStepTime-endStepTime".

**Example**  TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));  
**Usage:** double t1=sg.dt(4); //grabs the fifth time  
TimeTrain<UniformTimeTrain>::iterator myIt(sg);  
t1=myIt.dt(); //grabs the first element (which is 0.1)  
++myIt;  
t1=myIt.dt(); //we are now at he second element in the list (which  
is still 0.1)

## TimeTrain::element extraction

**Function:** `double endStepTime(int i);  
double endStepTime(); --> FOR ITERATORS`

**Input:** `i-->the index from which to grab a time`

**Description:** `Returns the the end time step at index 'i'`

**Example** `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));  
double t1=sg.endStepTime(4); //grabs the SIXTH time  
TimeTrain<UniformTimeTrain>::iterator myIt(sg);  
t1=myIt.endStepTime(); //grabs the second element (here it would be  
0.1) as we have not iterated yet  
++myIt;  
t1=myIt.endStepTime(); //we are now at he THIRD element in the list  
(which is 0.2)`

## TimeTrain::element extraction

**Function:**      **double endTime();**

**Input:**        void

**Description:**    Returns the end time

**Example Usage:**  TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));  
                          double t1=sg.endTime(); //grabs the end time (here it is 1)

## TimeTrain::element extraction

**Function:**  **double step(int i);**  
**double step(); --> FOR ITERATORS**

**Input:**  i-->the index from which to grab a step

**Description:**  Returns the the 'sub step' size at index 'i' or at current iterator position.

**Example**  TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));

**Usage:** double t1=sg.step(4); //grabs the fifth slot (here is 100)

TimeTrain<UniformTimeTrain>::iterator myIt(sg);

t1=myIt.step(); //grabs the first element (which is 100)

++myIt;

t1=myIt.step(); //we are now at he second element in the list  
(which is still 100)

## TimeTrain::element extraction

**Function:**  **double stepsize(int i);**  
 **double stepsize(); --> FOR ITERATORS**

**Input:**  i-->the index from which to grab a time

**Description:**  Returns the the 'sub step' TIME size, "dt()/step()", at the current iterator position

**Example**  TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));  
**Usage:**  double t1=sg.stepsize(4); //grabs the fifth slot (here is 1e-3)  
TimeTrain<UniformTimeTrain>::iterator myIt(sg);  
t1=myIt.stepsize(); //grabs the first element (which is 1e-3)  
++myIt;  
t1=myIt.stepsize(); //we are now at he second element in the list  
(which is still 1e-3)

## TimeTrain::element extraction

**Function:**  `double t(int i);`  
`double t(); --> FOR ITERATORS`

**Input:**  `i-->the index from which to grab a time`

**Description:**  `returns the time at the current iterator position or current index`

**Example**

**Usage:**   
`double t1=sg.t(2); //grabs the third element (here it would be 0.2)`  
`TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`  
`TimeTrain<UniformTimeTrain>::iterator myIt(sg);`  
`double t1=myIt.t(); //grabs the First element (here it would be 0)`  
`as we have not iterated yet`  
`++myIt;`  
`t1=myIt.t(); //we are now at he second element in the list (which`  
`is 0.1)`

## TimeTrain::assignments

**Function:**  **void addStep();**

**Input:**  none

**Description:**  Adds an extra element to the list...that new element will be (EndTime+Dt).  
The new end time them becomes OldEndTime+dt.  
The size of the list increases by 1.

**Example Usage:** 

```
UniformTimeEngine myE(0,1,10,100);
TimeTrain<UniformTimeTrain> sg(myE);
sg.addStep(); //end time is 1.1 and is '11' in length
```

## TimeTrain::assignments

**Function:**  **void dropStep();**

**Input:**  none

**Description:**  Drops the last step in the list.  
New End time becomes OldEndTime-dt.  
List decreases in size by 1.

**Example Usage:**  `UniforTimeEngine myE(0,1,10,100);  
TimeTrain<UniformTimeTrain> sg(myE);  
sg.dropStep(); //end time is 0.9 and is '9' in length`

## TimeTrain::assignments

**Function:** `TimeTrain &operator=(const TimeTrain<NewEng> &rhs);`

**Input:** `rhs-->` A Engine WITH THE SAME Time ENGINE! OR it can be a different engine IF the 'Engine\_t::operator=(NewEng)' is defined

**Description:** `Assigns one TimeTrain from another.`

The 'newEng' of the 'rhs' must be the same as the 'lhs' unless the `Engine_t::operator=(newEng)` is defined.

**Example** `UniforTimeEngine myE(0,1,10,100);`

**Usage:** `ConstUniformTimeEngine<256> meCE(0,10,1);`

`TimeTrain<UniformTimeTrain> sg(myE);`

`TimeTrain<ConstUniformTimeTrain<256> > sgC(meCE);`

`sg=sgC; //This will work`

`sgC=sg; //This will fail ConstEng=UniEng NOT defined`

## TimeTrain::assignments

**Function:** `TimeTrain &operator=(const EngType &rhs);`

**Input:** `rhs-->` A time engine WITH THE SAME Time ENGINE! OR it can be a different engine IF the 'Engine\_t operator=(EngType)' is defined

**Description:** `Assigns TimeTrain from an Engine Type....` THe 'newEng' of the 'rhs' must be the same as the 'lhs' unless the `Engine_t::operator=(EngType)` is defined.  
The time list is then initialized upon this assignment.

**Example** `UniforTimeEngine myE(0,1,10,100);`

**Usage:** `ConstUniformTimeEngine<256> meCE(0,10,1);`

`TimeTrain<UniformTimeTrain> sg;`

`TimeTrain<ConstUniformTimeTrain<256> > sgC;`

`sg=myE; //This will work`

`sg=meCE; //this will work too`

`sgC=myE; //This will fail ConstEng=UniEng NOT defined`

`sgC=meCE; //this will work`

## TimeTrain::assignments

**Function:** □ **void setBeginTime(double newbt);**

**Input:** □ newbt-->sets the beinging time of the list

**Description:** □ Alters the beinging time of the time list.  
CANNOT be bigger then the end time.

**Example Usage:** □ :: UniforTimeEngine myE(0,1,10,100);  
TimeTrain<UniformTimeTrain> sg(myE);  
sg.setBeginTime(3); //error bigger then the 'end time'  
sg.setEndTime(4); //set the end time  
sg.setBeginTime(3); //okay

## TimeTrain::assignments

- Function:**
  - **void setEndTime(double NewEndT);**
- Input:**
  - NewEndT-->sets the End time of the list
- Description:**
  - Alters the end time of the time list.
  - CANNOT be smaller then the Begining time.
- Example Usage:**
  - ```
UniforTimeEngine myE(0,1,10,100);
TimeTrain<UniformTimeTrain> sg(myE);
sg.setBeginTime(3); //error bigger then the 'end time'
sg.setEndTime(4); //set the end time
sg.setBeginTime(3); //okay
```

TimeTrain::assignments

Function: `void SetEngine(const Engine_t &rhs);`

Input: `rhs-->` A Time ENGINE of the same Type with an 'operator='() within the engine class....you can use this to convert between engine types IF you have defined an 'operator='() inside the engine the CAN convert between different engines

Description: `Assigns the Time Engine to the 'rhs' and reinitializes the Time lists.`

Example `UniforTimeEngine myE(0,1,10,100);`

Usage: `ConstUniformTimeEngine<256> meCE(0,10,1);`

```
TimeTrain<UniformTimeTrain> sg;
sg.SetEngine(myE); //sets the engine after the declaration
sg.SetEngine(meCE); //this will convert the ConstEng to an Uniform
Eng
TimeTrain<ConstUniformTimeTrain<256> > sgN;
sgN.SetEngine(meCE); //will set the Const Eng
sgN.SetEngine(myE); //This will fail ConstEng=UniEng NOT defined
```

TimeTrain::assignments

Function: `void setStep(int newStep);`

Input: `newStep`-->the NEW SIZE of the time train

Description: `Makes the Time train longer (or shorter) in the size, but keeps the begining and end Time the same.`

Example `UniforTimeEngine myE(0,1,10,100);`

Usage: `TimeTrain<UniformTimeTrain> sg(myE);`

```
sg.setStep(300); //this is now a 300 point time train going from 0-->1
```

TimeTrain::assignments

- Function:** **void setSubStep(int newSubStep);**
- Input:** newSubStep-->the NEW SUB Step Size of the time train
- Description:** Alters the SubStep size of the time list
- Example Usage:** `UniformTimeEngine myE(0,1,10,100);
TimeTrain<UniformTimeTrain> sg(myE);
sg.setSubStep(300); //subStep is now 300 (it was 100)`

TimeTrain::IO

Function: **void display(ostream &out);**
 void display();

Input: out--> an output stream (a file, a console, whatever)
VOID--> the void argument AUTO displays to the console

Description: This is the 'pretty' print option DO NOT USE THIS TO SAVE DATA TO BE READ LATTER...use writeASCII or write...
Prints the points valid to the time train to "out" in the following format ASCII

"step: <i> beginTime: <beingStepTime> endtime: <endsteptime> substep: <step>"

The void argument AUTO displays to the console.

Example TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));
Usage: ofstream out("myfile"); //an ofstream
sg.display(out);
//prints a list that looks like
//step: 0 beginTime: 0 endTime: 0.1 substep: 100
//step: 1 beginTime: 0.1 endTime: 0.2 substep: 100
//....
//step: 9 beginTime: 0.9 endTime: 1 substep: 100

TimeTrain::IO

Function: `ostream &operator<<(ostream &out, TimeTrain &toPrint);`

Input: `out-->` an ostream output buffer

`toPrint-->` The Timetrain you wish to print

Description: `The` is the 'Pretty Print' operator (uses 'display()'').

Prints the points valid to the time train to "out" in the following format ASCII.

"step: {i} beginTime: {beingStepTime} endTime: {endStepTime} substep: {step}"

Example Usage: `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10, 100));`

```
cout<<sg<<endl;
//prints a list that looks like to the console
//step: 0 beginTime: 0 endTime: 0.1 substep: 100
//step: 1 beginTime: 0.1 endTime: 0.2 substep: 100
//....
//step: 9 beginTime: 0.9 endTime: 1 substep: 100
```

TimeTrain::IO

Function: `ofstream &operator<<(ofstream &out, TimeTrain &tostream);`

Input: `out-->` an ofstream output ASCII FILE buffer
`tostream-->` The Timetrain you wish to print

Description: `The Overload of this operator..the ofstream is concidered to be an ASCII FILE and so it follows the "writeASCII" rules.`

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
 `ofstream OUT("outf"); //open ASCII file for writing`
 `OUT<<sg; //write to a ASCII file (write(OUT) should write the same`
 `thing also)`
 `TimeTrain<ConstUniformTimeEngine<20> >`
 `sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
 `OUT<<sgC; //write another TimeTrain in the SAME file`
 `OUT.close(); //close the file so we can open it in read mode`

 `//now we want to load up the written data...the order of the reads`
 `must be the same as the order of the writes..`
 `ifstream INF("outf");`
 `TimeTrain<UniformTimeEngine> rU;`
 `INF>>rU; //read the Uniform one...this is the same As`
 `'readASCII(ifstream)'`
 `TimeTrain<ConstUniformTimeEngine<20 > > rC;`
 `INF>>rC; //read the Const one..this is the same as 'read(ifstream)'`

TimeTrain::IO

Function: `fstream &operator<<(fstream &out, TimeTrain &toprint);`

Input: `out-->` an fstream output BINARY FILE buffer
`toprint-->` The Timetrain you wish to print

Description: `out-->` The Overload of this operator..the fstream is concidered to be an BINARY FILE and so it follows the "write" rules.

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
 `fstream OUT("outf", ios::binary | ios::out); //open BINARY file for writing`
 `OUT<<sg; //write to a binary file (write(OUT) should write the same thing also)`
 `TimeTrain<ConstUniformTimeEngine<20> >`
 `sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
 `OUT<<sgC; //write another TimeTrain in the SAME file`
 `OUT.close(); //close the file so we can open it in read mode`

 `//now we want to load up the written data...the order of the reads must be the same as the order of the writes..`
 `fstream INF("outf", ios::binary|ios::in);`
 `TimeTrain<UniformTimeEngine> rU;`
 `INF>>rU; //read the Uniform one...this is the same As 'read(ifstream)'`
 `TimeTrain<ConstUniformTimeEngine<20 > > rC;`
 `INF>>rC; //read the Const one..this is the same as 'read(ifstream)'`

TimeTrain::IO

Function: `ifstream &operator >> (ifstream &InFile, TimeTrain &toRead);`

Input: `InFile-->` an ofstream INPUT ASCII FILE buffer
`toread-->` The Timetrain you wish to print

Description: `□` The Overload of this operator...the ifstream is concidered to be an ASCII FILE and so it follows the "readASCII" rules.

Example `□ TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`

Usage: `//initialized once
ofstream OUT("outf"); //open ASCII file for writing
OUT<<sg; //write to a ASCII file (write(OUT) should write the same
thing also)`

`TimeTrain<ConstUniformTimeEngine<20> >
sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once
OUT<<sgC; //write another TimeTrain in the SAME file
OUT.close(); //close the file so we can open it in read mode`

`//now we want to load up the written data...the order of the reads
must be the same as the order of the writes..`

`ifstream INF("outf");
TimeTrain<UniformTimeEngine> rU;
INF>>rU; //read the Uniform one...this is the same As
'readASCII(ifstream)'
TimeTrain<ConstUniformTimeEngine<20 > > rC;
INF>>rC; //read the Const one..this is the same as 'read(ifstream)'`

TimeTrain::IO

Function: `fstream &operator >> (fstream &InFile, TimeTrain &toRead);`

Input: `InFile-->` an fstream INput BINARY FILE buffer
`toRead-->` The TimeTrain you wish to read

Description: `■` The Overload of this operator..the fstream is concidered to be an BINARY FILE and so it follows the "read" rules.

Example `■ TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
`fstream OUT("outf", ios::binary | ios::out); //open ASCII file for writing`
`OUT<<sg; //write to a binary file (write(OUT) should write the same thing also)`
`TimeTrain<ConstUniformTimeEngine<20> >`
`sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
`OUT<<sgC; //write another TimeTrain in the SAME file`
`OUT.close(); //close the file so we can open it in read mode`

`//now we want to load up the written data...the order of the reads must be the same as the order of the writes..`
`fstream INF("outf", ios::binary|ios::in);`
`TimeTrain<UniformTimeEngine> rU;`
`INF>>rU; //read the Uniform one...this is the same As 'read(ifstream)'`
`TimeTrain<ConstUniformTimeEngine<20 > > rC;`
`INF>>rC; //read the Const one..this is the same as 'read(ifstream)'`

TimeTrain::IO

Function: `void print(ostream &out);`

Input: `out-->` an output stream (a file, a console, etc.)

Description: `This another name for the "writeASCII" function...but this will print the data to any "ostream" which is any outputs stream you can think of.`

Example `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`

Usage: `sg.print(cout); //print the 'writeASCII' contents to the console`

TimeTrain::IO

Function: `void printFileTimEngine(ostream &out);`

Input: `out-->` an output stream (a file, a console, etc)

Description: `Prints the points valid to the time train that can be read by the "FileTimeTrain" Engine..in the following ASCII format {time} {substep}.`

Example `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`

`ofstream out("myfile"); //an ofstream
sg.printFileTimeTrain(out);
//prints a list that looks like to the console
//0 100
//0.1 100
//...
//1 100`

Usage:

TimeTrain::IO

Function:

- `void read();`
- `void read(const char *in);`
- `void read(string in);`

Input:

- `in-->` A new file name string

Description:

- These are for FileTimeEngine Only!

For FileTimeTrain it reads a valid 2 column format

{time} {substep}

If will pass this read string (or function) on to ANY engine you create with a read option

If the 'void' read...it will ReRead the the exisiting stored file name you gave it at 'construction'...if the 'string' input it will read the New file name

Example

- `TimeTrain<FileTimeEngine> sg(FileTimeTrain("myfile"));`

Usage:

- `//initialized once`

- `sg.read("myNewfile"); //reinistialized use the new file name`

TimeTrain::IO

Function: `int read(fstream &ReadFile);`

Input: `ReadFile-->` A binary file opened using 'fstream' with the ios options as.. "ios::binary | ios::in "

Description: `Reads from a binary file THE next instance of the Engine Type saved in the file.`
`Returns '1' if it worked, '0' if it failed .`

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
 `fstream OUT("outf", ios::binary|ios::out); //open binary file for writing`
 `sg.write(OUT); //write to a binary file`
 `TimeTrain<ConstUniformTimeEngine<20> >`
 `sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
 `sgC.write(OUT); //write another TimeTrain in the SAME file`
 `OUT.close(); //close the file so we can open it in read mode`

TimeTrain::IO

Function: `int read(ifstream &ReadFile);
int readASCII(ifstream &ReadFile);`

Input: `ReadFile-->` A binary file opened using 'ifstream' this ASSUMES an ASCII file reading

Description: `The overloading File Stream types...from the 'read(fstream)' above
Reads from a ASCII file THE next instance of the Engine Type saved in the file.
Returns '1' if it worked, '0' if it failed.`

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));
//initialized once`

`ofstream OUT("outf"); //open ASCII file for writing
sg.writeASCII(OUT); //write to a ASCII file (write(OUT) should
write the same thing also)`

`TimeTrain<ConstUniformTimeEngine<20> >
sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once
sgC.writeASCII(OUT); //write another TimeTrain in the SAME file
OUT.close(); //close the file so we can open it in read mode`

`//now we want to load up the written data...the order of the reads
must be the same as the order of the writes..`

`ifstream INF("outf");
TimeTrain<UniformTimeEngine> rU;
rU.read(INF); //read the Uniform one...this is the same As
'readASCII(ifstream)'
TimeTrain<ConstUniformTimeEngine<20 > > rC;
rC.readASCII(INF); //read the Const one..this is the same as
'read(ifstream)'`

TimeTrain::IO

Function: `int write(fstream &OutFile);`

Input: `OutFile-->` A binary file opened using 'fstream' with the ios options as.. "ios::binary | ios::in "

Description: `Writes from a binary file THE next instance of the Engine Type saved in the file.`
`Returns '1' if it worked, '0' if it failed.`

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
 `fstream OUT("outf", ios::binary|ios::out); //open binary file for writing`
 `sg.write(OUT); //write to a binary file`
 `TimeTrain<ConstUniformTimeEngine<20> >`
 `sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
 `sgC.write(OUT); //write another TimeTrain in the SAME file`
 `OUT.close(); //close the file so we can open it in read mode`

 `//now we want to load up the written data...the order of the reads must be the same as the order of the writes..`
 `fstream INF("outf", ios::binary|ios::in);`
 `TimeTrain<UniformTimeEngine> rU;`
 `rU.read(INF); //read the Uniform one`
 `TimeTrain<ConstUniformTimeEngine<20 > > rC;`
 `rC.read(INF); //read the Const one`

TimeTrain::IO

Function: `int write(ofstream &WFile);`
`int writeASCII(ofstream &WFile);`

Input: WFile--> A ASCII file opened using 'ofstream' this ASSUMES an ASCII file reading.

Description: Note the overloading File Stream types...from the 'write(fstream).'
Reads from a ASCII file THE next instance of the Engine Type saved in the file.
Rturns '1' if it worked, '0' if it failed.

Example `TimeTrain<UniformTimeEngine> sg(UniformTimeEngine(0,1,10,100));`
Usage: `//initialized once`
`ofstream OUT("outf"); //open ASCII file for writing`
`sg.writeASCII(OUT); //write to a ASCII file (write(OUT) should`
`write the same thing also)`
`TimeTrain<ConstUniformTimeEngine<20> >`
`sgC(ConstUniformTimeEngine<20>(0,1,100)); //initialized once`
`sgC.write(OUT); //write another TimeTrain in the SAME file`
`OUT.close(); //close the file so we can open it in read mode`

`//now we want to load up the written data...the order of the reads`
`must be the same as the order of the writes..`
`ifstream INF("outf");`
`TimeTrain<UniformTimeEngine> rU;`
`rU.read(INF); //read the Uniform one...this is the same As`
`'readASCII(ifstream)'`
`TimeTrain<ConstUniformTimeEngine<20>> rC;`
`rC.readASCII(INF); //read the Const one..this is the same as`
`'read(ifstream)'`

TimeTrain::other

Function: `int size();`

Input: none

Description: returns the total number of elements in the time train

Example Usage: `TimeTrain<UniformTimeTrain> sg(UniformTimeTrain(0, 1, 10,100));`
`int s=sg.size(); //the size here is 10;`

- Here is where the speed arises, and it is also here you need to be careful about which looping method you use. There are two basic types. One is meant for hyper Speed, but it has certain restrictions...the other is the more basic but is slower...
- **Iterator:: TimeTrain::iterator**
 - this loops over time train such that it loops from the first time to the Last-1 time
 - Time access, step sizes, and 'dt' times are obtained using the below functions
 - **beginStepTime()**--> returns the current begining time
 - **endStepTime()**-->return the time after the begining time
 - **t()**--> same as beginStepTime()
 - **step()**--> Number of fictitious steps between beginStepTime() and endStepTime()
 - **stepsize()**--> this (endStepTime()-beginStepTime())/step()
 - Iterations can occur backwards OR forwards
 - **operator++()**--> the foward iterator
 - **operator--()**--> the backwards iterator
 - If you wish to reuse the iterator AFTER you have gone foward all the way you can use the reset function
 - **reset()**--> resets the iterator back to the TOP most position
 - There is an extra option that allows you loop the iterations before they quit a loop
 - **setLoops(int loops)**--> sets the number of times to REPEAT the entire train before exiting a loop...the default is only 1...you can loop just once
- Below are some examples....

```
//a time train that is [0,0.1,0.2,0.3,0.4,0.5...1] with 'sub step sizes' of 100
TimeTrain<UniformTimeEngine> myT(0, 1, 10, 100);
TimeTrain<UniformTimeEngine>::iterator myIt(myT); //sets up an iterator to loop throught the time train

//this will output the list in foward order
// 0 0.1 100
// 0.1 0.2 100
//...
// 0.9 1 100
while(myIt){
    cout<<myIt.beginStepTime()<<" "<<myIt.endStepTime()<<" "<<myIt.step()<<endl;
    ++myIt; //advance the iterator
}
//There is a '--' operator that would take from the end to the begining
//if you needed to...so IF i DO NOT reset the iterator, i can go backwards

//use this to reset the iterator if you want to reuse it in the forward direction again
// myIt.reset();

//this loop will print out the list backwards
// 0.9 1 100
// 0.8 0.9 100
//...
// 0 0.1 100
while(myIt){
    cout<<myIt.beginStepTime()<<" "<<myIt.endStepTime()<<" "<<myIt.step()<<endl;
    --myIt; //decrement the iterator
}

//now that the iterator is back at the begining i will make it loop more then once
myIt.setLoops(3);
while(myIt){
    cout<<myIt.beginStepTime()<<" "<<myIt.endStepTime()<<" "<<myIt.step()<<endl;
    ++myIt; //decrement the iterator
}
//this will print out the same thing 3 times.

//prints the same output as the first iteration loop above
//but it will be slower AND there is no Bounds checking
//So you may Crash the program...
for(int i=0;i<myT.size()-1;++i){
    cout<<myT.beginStepTime(i)<<" "<<myT.endStepTime(i)<<" "<<myT.step()<<endl;
}
```


--Global FunctionsStencils...Finite differences on Vectors

- 1st order::backwards
- 1st order::central
- 1st order::forward
- 2nd order::backwards
- 2nd order::central
- 2nd order::forward
- 3rd order::backwards
- 3rd order::central
- 3rd order::foward
- 4th order::backwards
- 4th order::central
- 4th order::foward

Here you will find a body of functions designed to apply Stencils (or finite-difference operators) to vectors (i.e. 1D systems). To see the implementation for Grids look to Stencils-Grids.

Stencil operators are designed to implement numerical derivatives in a fast manner. The naming scheme for the operators is simple

Derivative{F,B,"}_{{depth}}_{{order}}{n}

{F,B,"} --> if 'F' use forward differences, if 'B' use backwards differences, if "(nonthing)" use central differences

{depth}-->which derivative (i.e. first, second, third, forth) it is a number from 1..4.

{order}--> the approximation order of the derivative

{n}--> if present it will NORMALIZE the derivative

Stencils-Vector::global function

Function:

- `Vector DerivativeB_1_1(Vector in)`
- `Vector DerivativeB_1_1n(Vector in)`
- `Vector DerivativeB_1_2(Vector in)`
- `Vector DerivativeB_1_2n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the first derivatives for using the backards difference calculation.

`1_1 --> v[i]= v[0]-v[i-1]`
`1_1n --> v[i]= v[0]-v[i-1]`
`1_2 --> v[i]= v[i-2]+3*v[0]-4*v[i-1]`
`1_2n --> v[i]= (v[i-2]+3*v[0]-4*v[i-1])/2`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0 , 5 , 1)), loo;`
`//loo=[0 2 2 2 2]`
`loo=DerivativeB_1_1(moo);`
`//note::: the last point is not effected`

Stencils-Vector::global function

Function:

- `Vector Derivative_1_2(Vector in)`
- `Vector Derivative_1_2n(Vector in)`
- `Vector Derivative_1_4(Vector in)`
- `Vector Derivative_1_4n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the first derivatives for using the 'Central' difference calculation.

The stencils are

`1_2 --> v(i)=v(i+1)-v(i-1);`
`1_2n --> v(i)=(v(i+1)-v(i-1))/2;`
`1_4 --> v(i)=v(i-2)-8.0*v(i-1)+8.0*v(i+1)-v(i+2);`
`1_4n --> v(i)=(v(i-2)-8.0*v(i-1)+8.0*v(i+1)-v(i+2))/12;`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0 , 5 , 1)),loo;`
`//loo=[0 2 2 2 4]`
`loo=Derivative_1_2(moo);`
`//note:: the first and last point are not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeF_1_1(Vector in)`
- `Vector DerivativeF_1_1n(Vector in)`
- `Vector DerivativeF_1_2(Vector in)`
- `Vector DerivativeF_1_2n(Vector in)`

Input:

- `in` --> a vector

Description:

- These are the first derivatives for using the foward difference calculation.

```
1_1 --> v[i]=v[i+1]-v[i];
1_1n --> v[i]=v[i+1]-v[i];
1_2 --> v[i]=4*v[i+1]-3*v[i]-v[i+2];
1_2n --> v[i]=(4*v[i+1]-3*v[i]-v[i+2])/2;
```

Example Usage:

- `//moo=[0 1 2 3 4]`
- `Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
- `//loo=[2 2 2 2 4]`
- `loo=DerivativeF_1_1(moo);`
- `//note:: the last point is not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeB_2_1(Vector in)`
- `Vector DerivativeB_2_1n(Vector in)`
- `Vector DerivativeB_2_2(Vector in)`
- `Vector DerivativeB_2_2n(Vector in)`

Input:

- `in` --> a vector

Description:

- These are the first derivatives for using the backards difference calculation.

`2_1 --> v[i]=v[0]+v[i-2]-2*v[i-1]`
`2_1n --> v[i]=v[0]+v[i-2]-2*v[i-1]`
`2_2 --> 2*v[0]+4*v[i-2]-5*v[i-1]-v[i-3];`
`2_2n --> 2*v[0]+4*v[i-2]-5*v[i-1]-v[i-3];`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0 , 5 , 1)),loo;`
`//loo=[0 1 0 0 0]`
`loo=DerivativeB_2_1(moo);`
`//note::: the last point is not effected`

Stencils-Vector::global function

Function:

- `Vector Derivative_2_2(Vector in)`
- `Vector Derivative_2_2n(Vector in)`
- `Vector Derivative_2_4(Vector in)`
- `Vector Derivative_2_4n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the second derivatives for using the central difference calculation.

`2_2-->v(i)=v(i-1)-2.0*v(i)+v(i+1);`
`2_2n-->v(i)=v(i-1)-2.0*v(i)+v(i+1);`
`2_4-->v(i)=30.*v(i) + 16.*(v(i-1)+v(i+1))-(v(i-2)+v(i+2))`
`2_4n-->v(i)=(30.*v(i) + 16.*(v(i-1)+v(i+1))-(v(i-2)+v(i+2)))/12`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0 , 5 , 1)),loo;`
`//loo=[0 0 0 0 4]`
`loo=Derivative_2_2(moo);`
`//note::: the first and last point are not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeF_2_1(Vector in)`
- `Vector DerivativeF_2_1n(Vector in)`
- `Vector DerivativeF_2_2(Vector in)`
- `Vector DerivativeF_2_2n(Vector in)`

Input:

- `in` --> a vector

Description:

- These are the second derivatives for using the foward difference calculation.

`2_1 --> v[i]=v[i]+v[i+2]-2*v[i+1];`
`2_1n --> v[i]=v[i]+v[i+2]-2*v[i+1];`
`2_2 --> v[i]=2*v[i]+4*v[i+2]-5*v[i+1]-v[i+3];`
`2_2n --> v[i]=2*v[i]+4*v[i+2]-5*v[i+1]-v[i+3];`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
`//loo=[0 0 0 3 4]`
`loo=DerivativeF_2_1(moo);`
`//note::: the last 2 points are not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeB_3_1(Vector in)`
- `Vector DerivativeB_3_1n(Vector in)`
- `Vector DerivativeB_3_2(Vector in)`
- `Vector DerivativeB_3_2n(Vector in)`

Input:

- `in` --> a vector

Description:

- These are the first derivatives for using the backards difference calculation.

`3_1 --> v[i]=3*v[i-2]+v[i-1]+v[0]-v[i-3];`

`3_1n --> v[i]=3*v[i-2]+v[i-1]+v[0]-v[i-3];`

`3_2 --> v[i]=5*v[i] - 18*v[i-1] + 24*v[i-2] -14*v[i-3] + 3*v[i-4];`

`3_2n --> v[i]=(5*v[i] - 18*v[i-1] + 24*v[i-2] -14*v[i-3] + 3*v[i-4])/2;`

Example Usage:

- `//moo=[0 1 2 3 4]`
- `Vector<double> moo(Spread<double>(0 , 5 , 1)) , loo;`
- `//loo=[0 1 2 0 0]`
- `loo=DerivativeB_3_1(moo);`
- `//note::: the last point is not effected`

Stencils-Vector::global function

Function:

- `Vector Derivative_3_2(Vector in)`
- `Vector Derivative_3_2n(Vector in)`
- `Vector Derivative_3_4(Vector in)`
- `Vector Derivative_3_4n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the third derivatives for using the central difference calculation.

`3_2--> v(i)=2*v(i-1)-v(v-2)-2*v(i+1)+v(i+2);`
`3_2n--> v(i)=(2*v(i-1)-v(v-2)-2*v(i+1)+v(i+2))/2;`
`3_4 --> v(i)=v(i-3) - 8*v(i-2) +13*v(i-1)`
`-13.*v(i+1)+8.*v(i+2)-v(i+3);`
`3_4n --> v(i)=(v(i-3) - 8*v(i-2) +13*v(i-1)`
`-13.*v(i+1)+8.*v(i+2)-v(i+3))/8;`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0 , 5 , 1)),loo;`
`//loo=[0 1 0 3 4]`
`loo=Derivative_3_2(moo);`
`//note:: the first and last point are not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeF_3_1(Vector in)`
- `Vector DerivativeF_3_1n(Vector in)`
- `Vector DerivativeF_3_2(Vector in)`
- `Vector DerivativeF_3_2n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the third derivatives for using the foward difference calculation.

`3_1 --> v[i]=3*v[i+1]-3*v[i+2]-v[0]+v[i+3]`
`3_1n --> v[i]=3*v[i+1]-3*v[i+2]-v[0]+v[i+3];`
`3_2 --> v[i]=18*v[i+1]-24*v[i+2]-5*v[0]+14*v[i+3]-3*v[i+4];`
`3_2n --> v[i]=18*v[i+1]-24*v[i+2]-5*v[0]+14*v[i+3]-3*v[i+4];`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
`//loo=[0 0 2 3 4]`
`loo=DerivativeF_3_1(moo);`
`//note::: the last 3 points are not effected`

Stencils-Vector::global function

Function:

- `Vector DerivativeB_4_1(Vector in)`
- `Vector DerivativeB_4_1n(Vector in)`
- `Vector DerivativeB_4_2(Vector in)`
- `Vector DerivativeB_4_2n(Vector in)`

Input:

- `in` --> a Vector

Description:

- These are the forth derivatives for using the backards difference calculation.

`4_1 --> v[i]=6*v[2]-4*v[i-1]-v[i]-4*v[i-3]+v[i-4];`
`4_1n --> v[i]=6*v[2]-4*v[i-1]-v[i]-4*v[i-3]+v[i-4];`
`4_2 --> v[i]=26*v[i-2]-14*v[i-1]-3*v[i]-24*v[i-3]+11*v[i-4]-2*v[i-5];`
`4_2n --> v[i]=26*v[i-2]-14*v[i-1]-3*v[i]-24*v[i-3]+11*v[i-4]-2*v[i-5];`

Example Usage:

- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
`//loo=[0 1 2 3 0]`
`loo=DerivativeB_4_1(moo);`
`//note::: the last point is not effected`

Stencils-Vector::global function

Function:

- `Vector Derivative_4_2(Vector in)`
- `Vector Derivative_4_2n(Vector in)`
- `Vector Derivative_4_4(Vector in)`
- `Vector Derivative_4_4n(Vector in)`

Input:

- `in--> a vector`

Description:

- These are the forth derivatives for using the central difference calculation.

```
4_2 --> v[i]=v[i-2]-4*v[i-1]+6*v[i+0]-4*v[i+1]+v[i+2];
4_2n--> v[i]=v[i-2]-4*v[i-1]+6*v[i+0]-4*v[i+1]+v[i+2];
4_4 --> v[i]=-v[i-3]+12*v[i-2]-39*v[i-1]
+56*A-39*v[i+1]+12*v[i+2]-v[i+3];
4_4n --> v[i]=(-v[i-3]+12*v[i-2]-39*v[i-1]
+56*A-39*v[i+1]+12*v[i+2]-v[i+3])/6;
```

Example Usage:

- `//moo=[0 1 2 3 4]`
- `Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
- `//loo=[0 1 0 3 4]`
- `loo=Derivative_4_2(moo);`
- `//note::: the first 2 and last 2 points are not effected`

Stencils-Vector::global function

- Function:**
- `Vector DerivativeF_4_1(Vector in)`
 - `Vector DerivativeF_4_1n(Vector in)`
 - `Vector DerivativeF_4_2(Vector in)`
 - `Vector DerivativeF_4_2n(Vector in)`
- Input:**
- `in--> a vector`
- Description:**
- These are the forth derivatives for using the foward difference calculation.
- `4_1--> v[i]=6.0*v[i+2]-4.0*v[i+1]-v[0]-4.0*v[i+3]+v[i+4];`
`4_1n --> v[i]=6.0*v[i+2]-4.0*v[i+1]-v[0]-4.0*v[i+3]+v[i+4];`
`4_2--> v[i]=26.0*v[i+2]-14.0*v[i+1]-3.0*v[0]-24.0*v[i+3]+11.0*v[i+4]-2.0*v[i+5];`
`4_2n--> v[i]=26.0*v[i+2]-14.0*v[i+1]-3.0*v[0]-24.0*v[i+3]+11.0*v[i+4]-2.0*v[i+5];`
- Example Usage:**
- `//moo=[0 1 2 3 4]`
`Vector<double> moo(Spread<double>(0, 5, 1)),loo;`
`//loo=[0 1 2 3 4]`
`loo=DerivativeF_4_2(moo);`
`//note:: the last 4 points are not effected`

--Global Functions[--- Derivative_1_2](#)[--- Derivative_2_2](#)[--- Derivative_3_2](#)[--- Derivative_4_2](#)[--- Gradient](#)[--- Laplace2D, Laplace2Dn](#)[--- Laplace3D, Laplace3Dn](#)**--Constructors**[--- StencilPrep](#)[--- StencilPrep](#)[--- StencilPrep](#)**--IO**[--- operator<<](#)[--- print](#)[--- printNeighbors](#)[--- printNextNeighbors](#)**--Other Functions**[--- calculate](#)[--- empty](#)[--- size](#)**Examples**[--- 1\) basic Stencils](#)Stencils...Finite differences using grids with data

Commonly, stencils are applied to data contained on a grid map. The difference between these stencils and the Stencil-Vector functions are the need for directionality (i.e. take a derivative along the x,y,z directions), and multi-dimensional stencils (Laplacians, Gradients, etc).

The functions are relatively easy to implement over rectangular grid shapes, where the boundaries are easily defined. However, the XYZshape class does not maintain the grid in a normal 3D array fashion because the shapes do not have to be rectangular. Thus the edges, faces, corners, etc are not well defined from the start. There is a need to determine the edges, neighbors, etc before a stencil can be applied. Because the grid is maintained as a Vector list, the data associated with a grid is also in a Vector format and the indices to the nearest-neighbors and next-nearest-neighbors are not ordered in the list. To solve these particular problems, a StencilPrep class is created to calculate the indices needed for these stencil operations.

The StencilPrep ‘calculation’ can be quite time consuming for large grids as it must check all grid points for a neighbor and next-nearest neighbor, but it only needs to be calculated ONCE for each grid. The calculation algorithm has been parallelized, but over relatively slow networks, it can actually take more time. Thus even if a program is run in parallel, you must explicitly define the StencilPrep object’s MPIcontroller in order to have it run in parallel. I recommend testing it with and without the controller present to see which one is faster.

The object itself contains many functions and has an iterator, but most of these functions you do not need to know unless you wish to use this object to perform something other than Stencils and Boundary conditions, only the basic necessary functions are described here. If you wish to know more, look to ‘src/stencils/stencil_prep.h’ for more information.

Once you have a StencilPrep object the stencils function require both the StencilPrep object AND the data vector.

Stencil-Grids::global function

Function:

- `Derivative_1_2(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Derivative_1_2n(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

- `void Derivative_1_2(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`
- `Derivative_1_2n(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`

Input:

- `outV`--> the vector you wish the calculated data to go to

`sp`--> a `StencilPrep` object

`inV`--> the input data `Vector`

`Direction`--> along which direction to apply the stencil(0=x, 1=y, 2=z)

`Slice`--> the direction within the Data Vector to apply the stencil (0=x, 1=y, 2=z)

Description:

- Takes the first derivative to second order, of a data vector, `inV`, whos data corresponds to the `StencilPrep`, `sp`. The result is placed in `outV`. Because the grid has three directions you can take the derivative along 3 directions 0=x, 1=y, 2=z.

If the Vector is full of coords, then there is an extra dimension (slice) which takes the derivative along direction (0,1,2) of the DATA.

There is a normalized version..`Derivative_1_2n`

And a forth order version..`Derivative_1_4`, `Derivative_1_4n`

Example

- see main example.

Usage:

Stencil-Grids::global function

Function:

- `Derivative_2_2(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Derivative_2_2n(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

- `void Derivative_2_2(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`
- `Derivative_2_2n(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`

Input:

- `outV`--> the vector you wish the calculated data to go to
- `sp`--> a `StencilPrep` object
- `inV`--> the input data `Vector`
- `Direction`--> along which direction to apply the stencil(0=x, 1=y, 2=z)
- `Slice`--> the direction within the Data `Vector` to apply the stencil (0=x, 1=y, 2=z)

Description:

- Takes the second derivative to second order, of a data `vector`, `inV`, whos data corresponds to the `StencilPrep`, `sp`. The result is placed in `outV`. Because the grid has three directions you can take the derivative along 3 directions 0=x, 1=y, 2=z.

If the `Vector` is full of coords, then there is an extra dimension (slice) which takes the derivative along direction (0,1,2) of the DATA.

There is a normalized version..`Derivative_2_2n`

And a forth order version..`Derivative_2_4`, `Derivative_2_4n`

Example
Usage:

- see main example

Stencil-Grids::global function

Function:

- `Derivative_3_2(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Derivative_3_2n(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

- `void Derivative_3_2(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`
- `Derivative_3_2n(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`

Input:

- `outV`--> the vector you wish the calculated data to go to
- `sp`--> a `StencilPrep` object
- `inV`--> the input data `Vector`
- `Direction`--> along which direction to apply the stencil(0=x, 1=y, 2=z)
- `Slice`--> the direction within the Data `Vector` to apply the stencil (0=x, 1=y, 2=z)

Description:

- Takes the third derivative to second order, of a data vector, `inV`, whos data corresponds to the `StencilPrep`, `sp`. The result is placed in `outV`. Because the grid has three directions you can take the derivative along 3 directions 0=x, 1=y, 2=z.

If the `Vector` is full of coords, then there is an extra dimension (slice) which takes the derivative along direction (0,1,2) of the DATA.

There is a normalized version..`Derivative_3_2n`

There is NO forth order version (we would need nextnextneighbor for that one).

Example

- see main example

Usage:

Stencil-Grids::global function

Function:

- `Derivative_4_2(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Derivative_4_2n(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

- `void Derivative_4_2(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`
- `Derivative_4_2n(Vector<coord<> &outV, StencilPrep &sp, Vector<coord<> &inV, int Direction, int Slice);`

Input:

- `outV`--> the vector you wish the calculated data to go to
- `sp`--> a `StencilPrep` object
- `inV`--> the input data `Vector`
- `Direction`--> along which direction to apply the stencil(0=x, 1=y, 2=z)
- `Slice`--> the direction within the Data `Vector` to apply the stencil (0=x, 1=y, 2=z)

Description:

- Takes the forth derivative to second order, of a data `vector`, `inV`, whos data corresponds to the `StencilPrep`, `sp`. The result is placed in `outV`. Because the grid has three directions you can take the derivative along 3 directions 0=x, 1=y, 2=z.

If the `Vector` is full of coords, then there is an extra dimension (slice) which takes the derivative along direction (0,1,2) of the DATA.

There is a normalized version..`Derivative_4_2n`

There is NO forth order version (we would need nextnextneighbor for that one).

Example

- see main example.

Usage:

Stencil-Grids::global function

Function:

- `Gradient(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Gradientn(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

Input:

- `outV`--> the vector you wish the calculated data to go to
- `sp`--> a `StencilPrep` object
- `inV`--> the input data `Vector`
- `Direction`--> along which direction to apply the stencil(0=x, 1=y, 2=z)

Description:

- Takes the Gradient along a specific direction (0,1,2)
If `Direction==0` the stencil takes the form
 $v[i] = v[i].nearest(-1,x) - v[i].nearest(1,x)$

- If `Direction==1` the stencil takes the form
 $v[i] = v[i].nearest(-1,y) - v[i].nearest(1,y)$

- If `Direction==2` the stencil takes the form
 $v[i] = v[i].nearest(-1,z) - v[i].nearest(1,z)$

The normalized version which normalizes with respect to the square of the grid spacing (`dx, dy, dz`).

Example

- see main example.

Usage:

Stencil-Grids::global function

Function:

- `Laplace2D(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`
- `Laplace2Dn(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV, int Direction);`

Input: □ outV--> the vector you wish the calculated data to go to

sp--> a StencilPrep object

inV--> the input data Vector

Direction--> along which direction to apply the stencil(0=x, 1=y, 2=z)

Description: □ Takes the 2D Laplacian along a particular Direction (x,y,z).

If Direction==0 then Lapacian2D takes a stencil of the form

$$v[i] = v[i].nearest(-1,y) -4.0*v[i]+ v[i].nearest(1,y)+\\ v[i].nearest(-1,z) + v[i].nearest(1,z);$$

If Direction==1 then Lapacian2D takes a stencil of the form

$$v[i] = v[i].nearest(-1,x) -4.0*v[i]+ v[i].nearest(1,x)+\\ v[i].nearest(-1,z) + v[i].nearest(1,z);$$

If Direction==2 then Lapacian2D takes a stencil of the form

$$v[i] = v[i].nearest(-1,x) -4.0*v[i]+ v[i].nearest(1,x)+\\ v[i].nearest(-1,y) + v[i].nearest(1,y);$$

The normalized version which normalizes with respect to the square of the grid spacing (dx, dy, dz).

Example

□ See main example.

Usage:

Stencil-Grids::global function

Function: `Laplace3D(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV);`
`Laplace3Dn(Vector<T> &outV, StencilPrep &sp, Vector<T> &inV);`

Input: outV--> the vector you wish the calculated data to go to
sp--> a StencilPrep object
inV--> the input data Vector

Description: Takes the 3D Laplacian along the entire grid.

```
v[i]=-6.0v[i]+  
v[i].nearest(-1,x) + v[i].nearest(1,x)+  
v[i].nearest(-1,y) + v[i].nearest(1,y)+  
v[i].nearest(-1,z) + v[i].nearest(1,z);
```

The normalized version which normalizes with respect to the square of the grid spacing (dx, dy, dz).

Example See main example.

Usage:

Stencil-Grids::constructor

Function: `StencilPrep<Shape_t>()`

Input: `void`

Description: The empty constructor. Requires the shape type as its template parameter.

Example Usage: `StencilPrep<XYZshape<XYZrect > > grid;`

Stencil-Grids::constructor

Function: `StencilPrep<Shape_t > stprep(Shape_t &grid, MPIcontroller &controller=MPIcontroller())`

Input: grid--> an XYZshape object
controller --> an MPIcontroller object (usually 'MPIworld')

Description: This is the basic constructor from an XYZshape. If you do not input anything into the MPIcontroller, then it will NOT calculate anything in parallel.

It automatically calculates the neighbor information.

Example `Grid<UniformGrid> basicgrid(min,max, dims);`

Usage: `XYZrect rect(min,max)`
`XYZshape<XYZrect > grid(basicgrid, rect);`
`//not in parallel`
`StencilPrep<XYZshape<XYZrect > > stprep(grid);`
`//in parallel`
`StencilPrep<XYZshape<XYZrect > > stprepP(grid, MPIworld);`

Stencil-Grids::constructor

Function: `template<class ShapeExpr>`
`StencilPrep<Shape_t > stprep(Shape_t &grid, 'ShapeExpr'`
`bounds,MPIcontroller &controller=MPIcontroller())`

Input: grid--> a Grid<UniformGrid> or XYZshape

bounds --> a shape expression

controller --> an MPIcontroller object (usually 'MPIworld')

Description: This is the basic constructor from an XYZshape or Grid<UniformGrid>, and the Shape Expression (i.e. XYZrect || XYZcylinder). If you do not input anything into the MPIcontroller, then it will NOT calculate anything in parallel.

It automatically calculates the neighbor information.

Example `Grid<UniformGrid> basicgrid(min,max, dims);`

`XYZrect rect(min,max);`

`XYZrect anotherRect(min2, max2);`

`XYZshape<XYZrect > grid(basicgrid, rect);`

`//not in parallel`

`StencilPrep<XYZshape<XYZrect > > stprep(grid,anotherRect && rect);`
`//in parallel`

`StencilPrep<XYZshape<XYZrect > > stprepP(grid,anotherRect && rect,`
`MPIworld);`

Stencil-Grids::IO

Function: `std::ostream &operator<<(std::ostream &oo, StencilPrep<Shape_t> &out)`

Input: `oo-->` an out stream
`out-->` a `StencilPrep` object

Description: Same as `'print(oo)'`.

Example `StencilPrep<Shape_t> stprep(grid);`

Usage: `cout<<stprep<<endl;`

Stencil-Grids::IO

Function: `void print(std::ostream &oo)`

Input: `oo-->` an ouput stream

Description: `Prints out the grid points in the object to the ostream.`

Example Usage:

```
Grid<UniformGrid> basicgrid(min,max, dims);
XYZrect rect(min,max)
XYZshape<XYZrect > grid(basicgrid, rect);
//not in parallel
StencilPrep<XYZshape<XYZrect > > stprep(grid);
stprep.print(cout);
```

Stencil-Grids::IO

Function:

□ `void printNeighbors(std::ostream &oo)`

Input:

□ `oo-->` an output stream

Description:

□ Prints out the neighbor data object points in the object to the ostream.

Example Usage:

□ `StencilPrep<Shape_t> stprep(grid);
stprep.printNeighbors(cout);`

Stencil-Grids::IO

Function: `void printNextNeighbors(std::ostream &oo)`

Input: `oo-->` an output stream

Description: `Prints out the next-neighbor data object points in the object to the ostream.`

Example Usage: `StencilPrep<Shape_t> stprep(grid);
stprep.printNextNeighbors(cout);`

Stencil-Grids::other

Function: `template<class ShapeExpr>`
`bool calculate(Shape_t &ins, ShapeExpr ine)`

Input: ins--> a Grid
ine --> a Shape Expression (like XYZrect, or XYZrect || XYZplaneXY)

Description: This is the master calculation function. You would need to call this function again anytime the Grid changes or you wish another Shape Expression to be used. This function can be time consuming for large grids.

Example `Grid<UniformGrid> basicgrid(min,max, dims);`

Usage: `XYZrect rect(min,max)`
`XYZshape<XYZrect > grid(basicgrid, rect);`
`//not in parallel`
`StencilPrep<XYZshape<XYZrect > > stprep(grid);`
`//in parallel`
`StencilPrep<XYZshape<XYZrect > > stprepP(grid, MPIworld);`
`//calculate using the shape`
`stprepP.calculate(grid, rect);`

Stencil-Grids::other

Function: **bool empty()**

Input: void

Description: Returns true if there are NO grid points in the object.

Example Usage: Grid<UniformGrid> basicgrid(min,max, dims);
 XYZrect rect(min,max)
 XYZshape<XYZrect > grid(basicgrid, rect);
 //not in parallel
 StencilPrep<XYZshape<XYZrect > > stprep(grid);
 //in parallel
 StencilPrep<XYZshape<XYZrect > > stprepP(grid, MPIworld);
 bool ee=stprepP.empty(); //false

Stencil-Grids::other

Function: `int size()`

Input: `void`

Description: `>Returns the size of the point in the stencil prep object. This can be different then the number of grid points because you could have used a different ShapeExpression then the one in the Grid.`

Example `Grid<UniformGrid> basicgrid(min,max, dims);`

Usage: `XYZrect rect(min,max)`
`XYZshape<XYZrect > grid(basicgrid, rect);`
`//not in parallel`
`StencilPrep<XYZshape<XYZrect > > stprep(grid);`
`//in parallel`
`StencilPrep<XYZshape<XYZrect > > stprepP(grid, MPIworld);`
`int mysiz=stprep.size();`

Performing various Stencil operators on grids and data.

Goes through the usual set up to calculate stencils over grid spaces.

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

int main(int argc, char **argv)
{
//some initial typedefs
    typedef XYZrect TheShape;
    typedef XYZshape<TheShape> TheGrid;

    typedef double GradType ;
    typedef double FieldType;
    coord<> mins(-1,-1,-1), maxs(1,1,1);
    coord<int> dims(5,5,5);

    Grid<UniformGrid> gg(mins, maxs, dims);

    TheShape tester(mins, maxs);
    TheGrid jj(gg,tester);

//set up our stencil object
    StencilPrep<TheGrid> sp(jj,tester);

//initialize some data vectors
    Vector<FieldType > moo(jj.size(),0);
    Vector<GradType > koo(moo.size(),0), koo2(koo), koo3(koo), koo4(koo), koo5(koo);
    TheGrid::iterator It(jj);

//fill our initial data with some function of the
// grid points (x+y^2+z^2)
    while(It){
        moo[It.curpos()]=It.x()+It.y()*It.y()+It.z()*It.z();
        ++It;
    }

//to make the directions clearer
    int x=0, y=1, z=2;

//take the gradient along the x direction
    Gradient(koo, sp, moo, x);
//take the first derivative along the x direction
    Derivative_1_2n(koo2, sp, moo,x);
//take the first derivative along the y direction
    Derivative_1_2n(koo3, sp, moo,y);
//take the Laplacian along the z direction
    Laplace2Dn(koo4, sp, moo,z);
//take the Laplacian of the entire grid
    Laplace3Dn(koo4, sp, moo);

//dump out the data
    ofstream oo1("grad"), oo2("derivX"),
        oo3("derivY"), oo4("lap2D"),
```

```
oo5( "lap3D" );

for( int i=0;i<koo.size();++i){
    oo1<<jj.Point(i)<<" "<<koo[i]<<endl;
    oo2<<jj.Point(i)<<" "<<koo2[i]<<endl;
    oo3<<jj.Point(i)<<" "<<koo3[i]<<endl;
    oo4<<jj.Point(i)<<" "<<koo4[i]<<endl;
    oo5<<jj.Point(i)<<" "<<koo5[i]<<endl;
}
}
```

--Public Vars[--- begin](#)[--- end](#)[--- eps](#)**--Constructors**[--- Integrate](#)[--- Integrate](#)**--Other Functions**[--- integrate](#)[--- integrate](#)[--- integrate](#)[--- operator\(\)](#)**Examples**[--- Integration](#)

```
template<class Func_T, class T=double>
class Integrate {...
```

This class simply acts as an integration engine for numerical integration of functions. It uses an adaptive Simpson method (W.Gander and W.Gautschi "Adaptive Quarature revisited" , 1998.) to perform the integration.

The template argument 'T' is the presision numeric type int, float, double, complex, etc. (default=double)

the class 'Func_T' should contain these functions and constructors

```
//the basic constructor
Func_T::Func_T()
//returns the value of the function at point 'x'
double Func_T::operator(double x);
```

Integrate::Public Vars

Function:

T begin

Input:

Description:

The start of the integration.

Example Usage:

`Integrate<Func_T> myI;`
`myI.begin=4;`

Integrate::Public Vars

Function:

T end

Input:

Description:

The ending value for the integration.

Example Usage:

```
 Integrate<Func_T> myI;  
myI.begin=4;  
myI.end=7;
```

Integrate::Public Vars

Function:

□ **T eps**

Input:

□

Description:

□ The relative error level desired

Example Usage:

□ `Integrate<Func_T> myI;`
`myI.begin=4;`
`myI.end=8;`
`myI.eps=1e-3;`

Integrate::constructor

Function: **Integrate<Func_T>()**

Input: void

Description: The empty constructor. Creates a new pointer to the Func_T (which will be deleted at destruction).
The default integration limits are 0..1.

Example class myF{

Usage: myF(){}
double operator(double x){ return x*x; }
};

Inegrate<myF> myInt;

Integrate::constructor

Function: `Integrate<Func_T, T>(Func_T &in, double begin=0, double end=1, double eps=1e-6)`

Input: in--> the function you wish to evaluate.
begin--> the begining x value
end --> the ending x value
eps--> the relative presision

Description: The basic constructor. Requires the function input. but the begining, ending, and presision values will assume the defaults if not preset.

Example `class myF{`

Usage: `myF(){}
double operator(double x){ return x*x; }
};

myF F;
Inegrate<myF> myInt(F, 3, 56);`

Integrate::other

Function:

T integrate();

Input:

void

Description:

Integrates the function from the default begin and end values.

Example Usage:

```
 Integrate<Func_T> myInt;  
myInt.begin=4;  
myInt.end=56;  
cout<<myInt.integrate()<<endl;
```

Integrate::other

Function:	<code>T integrate(T begin, T end);</code>
Input:	<ul style="list-style-type: none">□ begin --> the begining x value□ end --> the ending x value
Description:	<ul style="list-style-type: none">□ Integrates the function from begin to end.
Example Usage:	<ul style="list-style-type: none">□ <code>Integrate<Func_T> myInt;</code>□ <code>cout<<myInt.integrate(5, 56)<<endl;</code>

Integrate::other

Function: □ `Vector<T> integrate(Vector<T2> xlist);`
□ `Vector<T> integrate(Spread<T2> xlist);`
□ `Vector<T> integrate(Range xlist);`

Input: □ `xlist-->` a range of values (increasing) for integrate between.

Description: □ This will return a vector of integrated points. The size of the return vector will be 1 less than the length of the input time container.

Example □ `Integrate<Func_T> myInt;`

Usage: □ `Spread<double> xV(0, 4, 0.1);`
`cout<<myInt.integrate(xV)<<endl;`

Integrate::other

Function:

`T operator()()`
 `T operator()(T begin, T end)`

Input:

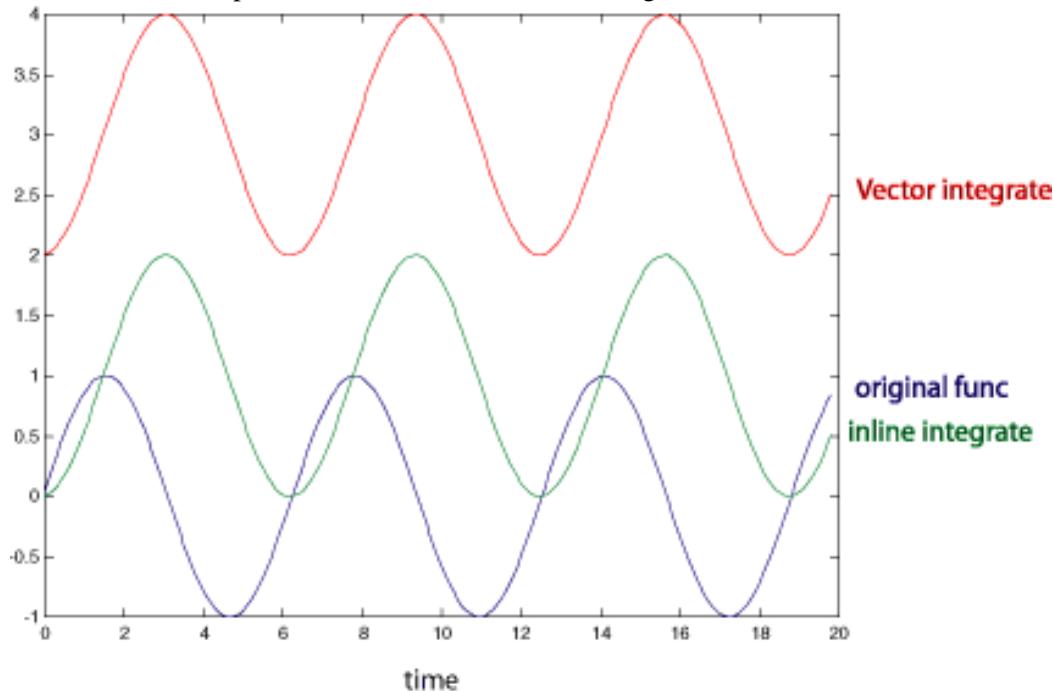
Description:

Same as 'integrate()'.
 Same as 'integrate(begin, end)'.

Example Usage:

A simple Integration usage example.

The code below ouputs the data shown in the below figure



Remeber that the integration is valid up to a constant (and that constant cannot be known without some other parameters). So the figure is showing the correct integration 'functional form' upto an addative constant. Also note that I added '2' to the vector data so it is visible.

```
#include "blochlib.h"
//the required 2 namespaces
using namespace BlochLib;
using namespace std;

//a function class to integrate (here it is sin(x))
//NOTE:: you need to define the "double operator(double)" function
class F
{
public:
    F(){}
    double operator()(double x)
    {   return sin(x);   }
};

int main()
{
    //declare the Integrate object first and then reuse it as nessessary
    Integrate<F> myint;
    ofstream out("out");

    //first out spread in time
    double dt=.1, a=0, b=20, integ=0;
    Vector<double> ts(Spread<double>(a,b,dt));
```

```
//we could integrate this way
//using a spread of values
Vector<double> intV=myint.integrate(ts);
int i=0;
//here we integrate F from t=0..20 in step sizes of dt=0.1

F myF; //so we can get the unintegrated values
while(i<ts.size()-1)
{
    //to Integrate from t1 to t2 simply use the "operator(t1, t2)" function
    //or use the function "integrate(t1, t2)"

    //integ+=myint(a, a+dt);
    //or supplementary
    integ+=myint.integrate(a, a+dt);

    //print "<time> <orig> <integ1> <intV>" to file
    //we add '2' to see the two integrations at the same time on
    // a graph
    out<<ts[i]<< " <<myF(ts[i]+dt/2.0)<< " <<integ<< " <<intV[i++]+2<<endl;
    a+=dt;
}
return 1;
}
```

--Constructors class Isotope{....

--- Isotope

--- Isotope

--- Isotope

--Element Extraction This class acts as a handy container for all the various bits of data associated with atomic isotope data...(masses, atomic numbers, lables, etc)

--- element

--- gamma

--- gammaGauss

--- HS

--- mass

--- momentum

--- name

--- number

--- qn, spin

--- relativeFrequency

--- reseptivity

--- symbol

--- weight

--Assignments

--- operator=

--- operator=

--Math

--- ==, !=

--IO

--- operator<<

Isotope::constructor

Function: **Isotope()**

Input: void

Description: empty constructor--> Sets up the default Isotope

Example Usage: Isotope moo;
 std::string sys=moo.symbol(); //returns "1H"

Isotope::constructor

Function: `Isotope(std::string symbol)`

Input: `symbol`--> the name of the atom...in the format "{atomic mass}{periodic label}" like "1H", "2H", "7Li"...etc

Description: `creates an isotope data set from the correct 'symbol'`

Example `Isotope moo("13C");`

Usage: `std::string sys=moo.symbol(); //returns "13C"`

Isotope::constructor

Function: **Isotope(const Isotope &cp)**

Input: cp--> an existing isotope to copy..

Description: creates a copy of the old isotope

Example Usage: `Isotope moo("27Al");`
 `Isotope loo(moo); //moo is copied into loo`

Isotope::element extraction

Function: **std::string element()**

Input: none

Description: Returns the element label.

Example Usage: `Isotope moo("1H");
cout<<moo.element(); //prints out 'H'`

Isotope::element extraction

Function: **double gamma()**

Input: none

Description: Returns the gamma factor (gyromagnetic ratio) for the nucleous in units of Hz(rad)/Telsa.

Example Usage: Isotope moo("1H");
cout<<moo.gamma(); //prints out '26.7519E+7'

Isotope::element extraction

Function: **double gammaGauss()**

Input: none

Description: Returns the gamma factor (gyromagnetic ratio) for the nucleous in units of Hz(rad)/Gauss.

Example Usage: `Isotope moo("1H");
cout<<moo.gammaGauss(); //prints out '26.7519E+3'`

Isotope::element extraction

Function: **int HS()**

Input: none

Description: Returns the hilbert space dimension for the spin (HS=2*Pspin quantum number}+1).
This is essentially the array size needed to accurately describe the spins behavior.

Example Usage:

```
Isotope moo( "1H" );
cout<<moo.HS(); //prints out '2'
```

Isotope::element extraction

Function: **double mass()**

Input: none

Description: Returns the mass in atomic Mass Units (amu).

Example Usage: Isotope moo("27Al");
cout<<moo.mass(); //prints out '27'

Isotope::element extraction

Function: `std::string momentum()`

Input: `none`

Description: `Returns the string form of 'qn().'`

Example Usage: `Isotope moo("1H");
cout<<moo.momentum(); //prints out '1/2'`

Isotope::element extraction

Function:	<input type="checkbox"/> std::string name()
Input:	<input type="checkbox"/> none
Description:	<input type="checkbox"/> Returns the full name of the isotope.
Example Usage:	<input type="checkbox"/> <code>Isotope moo("1H");</code> <code>cout<<moo.name(); //prints out 'Hydrogen'</code>

Isotope::element extraction

Function:

int number()

Input:

none

Description:

Returns the atomic number of the isotope.

Example Usage:

```
 Isotope moo( "27Al" );
cout<<moo.number(); //prints out '27'
```

Isotope::element extraction

Function:

- double qn();**
- double spin();**

Input:

- none

Description:

- Returns the spin quantum number of the isotope.

Example Usage:

- ```
Isotope moo("1H");
cout<<moo.qn(); //prints out '0.5'
cout<<moo.spin(); //prints our '0.5'
```

## Isotope::element extraction

**Function:**  **double relativeFrequency()**

**Input:**  none

**Description:**  Returns the relative frequency of the spin REALTIVE TO 1H.

**Example Usage:**  Isotope moo( "1H" );  
cout<<moo.relativeFrequency(); //prints out '1'  
Isotope loo("13C");  
cout<<loo.relativeFrequency(); //prints out '0.3207'

## Isotope::element extraction

**Function:**  **double reseptivity()**

Input:  none

Description:  Returns the relative sensitivity of the nucleous RELATIVE TO 13C.

Example  Isotope moo("1H");

Usage: cout<<moo.reseptivity(); //prints out '5.68E+3' (i.e. it has a signal 5.68E+3 times stronger then 13C)

## Isotope::element extraction

**Function:**  **std::string symbol()**

**Input:**  none

**Description:**  Returns the symbol name of the isotope.

**Example Usage:**  Isotope moo( "27Al" );  
cout<<moo.symbol(); //prints out '27Al'

## Isotope::element extraction

**Function:**  **double weight()**

**Input:**  none

**Description:**  Returns the mass in atomic Mass Units (amu) of the NATURAL abundance.

**Example Usage:**  Isotope moo("1H");  
cout<<moo.weight(); //prints out '1.008665'

## Isotope::assignments

**Function:** `Isotope &operator=(const Isotope &rhs);`

**Input:** `rhs-->` the item you wish to copy

**Description:** `Assigns one Isotope from another.`

**Example Usage:**

```
Isotope moo("27Al");
Isotope koo;
koo=moo;
```

## Isotope::assignments

**Function:** `Isotope &operator=(const std::string &sym);`

**Input:** `sym-->` A symbol string

**Description:** `Assigns the isotope from the symbol name.`

**Example Usage:** `Isotope moo;  
koo="27Al";`

## Isotope::Math

### Function:

- `bool operator==(const Isotope &rhs);`
- `bool operator!=(const Isotope &rhs)`

### Input:

- `rhs-->` another Isotope

### Description:

- Determines whether or not the rhs is the same isotope as the lhs.

### Example Usage:

- `Isotope moo("1H"), loo("13C");`
- `moo==loo; //would be false`
- `moo!=loo; //would be true`

## Isotope::IO

**Function:**  `ostream &operator<<( Isotope &iso)`

**Input:**  `iso-->` an isotope

**Description:**  `writes the string....`  
`'Isotope: {symbol}({momentum},{gamma})'`

**Example**  `Isotope moo( "1H");`

**Usage:**  `cout<<moo; // will write the string "Isotope: 1H(monmentum=1/2,  
gamma=2.6752E+8)"`

--Constructors template<int BPops=BPoptions::Density | BPoptions::HighField, class Offset\_T=double>  
--- BlochParams  
--- BlochParams  
--- BlochParams

---

## --Element Extraction

--- Bo

--- Mo

--- moles

--- temperature

## --Assignments

--- Bo

--- Mo

--- moles

--- operator=

--- operator=

--- operator=

--- temperature

## --Math

--- ==, !=

## --IO

--- operator<<

--- operator>>

--- print

--- read

--- write

## --Other Functions

--- calcMo

--- setInitialCondition

- This class acts as an extension to the 'Isotope' class and includes settings for the BASIC parameters used in solving the bloch
- BPops--> This is where you specify the "Type" of parameter it is...
  - **BPoptions::Density | BPoptions::HighField** --> Each parameter is considered a volume of spins which 'Bulk' properties...Equilibrium magnetization is given by The Curie Law and is All initially Mz (the HighField assumption)...You should use this is you are considering effects like Bulk Suseptability, or Radiation Damping, or the Demagnetizing Field...Diffusion is also treated differently in this case..the 'Dipole-Dipole' interaction HERE is meaningless and is replaced with the Demagnetizing Field
  - **BPoptions::Particle | BPoptions::HighField** --> Each parameter is a 'single spin' (so things like Radiation Damping, etc are invalid interactions)...This is used mainly to demonstrate and simulate the Dipole-Dipole interaction
  - **BPoptions::Density | BPoptions::ZeroField** -->Not Yet Ready
  - **BPoptions::Particle | BPoptions::ZeroField** --> Not Yet Ready
- **Offset\_T**--> The offset Type (a Double or coord )
  - **double** -->The standard high field along the z-hat direction assumption...because the Bfield along Z is so strong, the other directions are all truncated with respect to this axis so everything (Bo, Mo, offsets, T2, T1, etc) are all along this main axis...
  - **coord** --> "off axis high fields" where one can no longer assume the interactions are truncated along the z-axis...but an arbitrary axis relative to the lab frame. offsets, Bo, Mo, etc all become 3 vectors

## Bloch Parameters::constructor

**Function:**  **BlochParams()**

**Input:**  none

**Description:**  empty constructor--> Sets up the default Parameter  
The Default isotope is "1H"  
The Default Bo\_=4.7 tesla (if Offset\_T==coord<>, Bo\_=(0,0,4.7))  
The Default Mo\_=Curie Law (if Offset\_T==coord<>, Mo\_=(0,0, Mo), for Particle Mo\_=(0,0,1))  
The Default Temperature=300 K  
The Default moles=1

**Example Usage:**  BlochParam<> moo;  
std::string sys=moo.symbol(); //returns "1H"

## Bloch Parameters::constructor

**Function:** `blochParams(std::string symbol)`

**Input:** `symbol`--> the name of the atom...in the format "<atomic mass><periodic lable>" like "1H", "2H", "7Li"...etc

**Description:** `Creates an params data set from the correct 'symbol.'`

**Example** `BlochParams moo( "13C" );`

**Usage:** `std::string sys=moo.symbol(); //returns "13C"`

## Bloch Parameters::constructor

**Function:** `blochParams(const BlochParams &cp)`

**Input:** `cp-->` an existing isotope to copy..

**Description:** `Creates a copy of the old BlochParam.`

**Example Usage:** `BlochParams moo( "27Al" );  
BlochParams loo(moo); //moo is copied into loo`

## Bloch Parameters::element extraction

**Function:**  **Offset\_T &Bo( )**

**Input:**  none

**Description:**  Returns the Magnetic field on that spin (also can be used for assignment).

**Example Usage:**  BlochParams moo( "27Al" );  
moo.Bo( )=5;  
moo.calcMo();  
cout<<moo.Bo(); //prints out '5'

## Bloch Parameters::element extraction

**Function:**  **Offset\_T Mo()**

**Input:**  none

**Description:**  The initial total magnetization of the 'spin' at equilibrium inside a large magnetic field.

Each spin has the equilibrium magnetization given by the curie formula

$$Mo = (\gamma * hbar * \tanh(hbar * \pi * (Bo() * \gamma) / (2 * k_b * temperature()) * moles() * N_A * 1e3)) / 2.0$$

**Example Usage:**  BlochParams moo( "1H" );  
moo.Bo( )=4.7;  
moo.calcMo();  
cout<<moo.Mo(); //prints out something ~10E-4

## Bloch Parameters::element extraction

**Function:** `double &moles()`

**Input:** `none`

**Description:** `Returns the number of moles of that spin (also can be used for assignment).`

**Example Usage:**

```
BlochParams moo("27Al");
moo.Moles()=0.5;
moo.calcMo();
cout<<moo.moles(); //prints out '0.5'
```

## Bloch Parameters::element extraction

**Function:**  **double &temperature()**

**Input:**  void

**Description:**  Returns the Temperature of the spin IN KELVIN (also can be used for assignment).

**Example Usage:**  BlochParams moo( "27Al" );  
moo.temperature()=256.0;  
moo.calcMo();  
cout<<moo.temperature(); //prints out '256'

## Bloch Parameters::assignments

### Function:

□ **void Bo(Offset\_T newB)**

### Input:

□ newB--> the new magnetic field in Tesla

### Description:

□ Sets the Main Magnetic field of the spin.

### Example Usage:

□ `BlochParams moo( "27Al" );  
moo.Bo(2.0); //Bo is now 2.0 Telsa`

## Bloch Parameters::assignments

### Function:

□ **void Mo(Offset\_T newM)**

### Input:

□ newM--> the new equilibrium magnetization

### Description:

□ Sets the equilibrium magnetization of the spin.

### Example Usage:

□ `BlochParams<> moo( "27Al" );  
moo.Mo( 2.0e-5 ); //Mo is now 2.0e-5`

## Bloch Parameters::assignments

**Function:**       **void moles(double newM)**

**Input:**             newM--> the New moles

**Description:**         Sets the Moles of the spin.

**Example Usage:**     BlochParams moo( "27Al" );  
                        moo.Moles(2.0); //number of moles is now 2.0

## Bloch Parameters::assignments

**Function:**  `BlochParams &operator=(const BlochParams&rhs);`

**Input:**  `rhs-->` the item you wish to copy.

**Description:**  Assigns one BlochParams from another.

**Example Usage:**  `BlochParams moo( "27Al" );  
BlochParams koo;  
koo=moo;`

## Bloch Parameters::assignments

**Function:**       **BlochParams &operator=(const std::string &sym);**

**Input:**       sym--> A symbol string

**Description:**       Assigns the isotope from the symbol name.

**Example Usage:**      

```
BlochParams moo("27Al");
BlochParams koo;
koo="1H";
```

## Bloch Parameters::assignments

**Function:** `blochparams &operator=(const Isotope &sym);`

**Input:** `sym-->` An isotope object

**Description:** `Assigns the isotope from another isotope.`

**Example Usage:**

```
Isotope moo("13C");
BlochParams koo;
koo=moo;
```

## Bloch Parameters::assignments

**Function:** `void temperature(double newT)`

**Input:** `newT--> the New Temperature in Kelvin`

**Description:** `Sets the Temperature of the spin.`

**Example Usage:** `BlochParams moo( "27Al" );  
moo.Temperature(245.0); //Temperature is now 245 K`

## Bloch Parameters::Math

### Function:

- `bool operator==(const BlochParams &rhs);`
- `bool operator!=(const BlochParams &rhs);`

### Input:

- `rhs-->` another BlochParams

### Description:

- Determines whether or not the rhs is the same isotope as the lhs.

### Example Usage:

- `BlochParams moo("1H"), loo("13C");`
- `moo!=loo; //would be true`

## Bloch Parameters::IO

**Function:** `o ostream &operator<<(BlochParams &bp)`

**Input:** `bp-->` the BlochParams you wish to output

**Description:** `o` writes the string....

"BlochParam: offset={offset}, T1={T1}, T2={T2}, moles={Moles}, Spin={symbol},  
momentum={Momentum}, gamma={gamma}"

**Example** `o BlochParams moo( "1H" );`

**Usage:** `cout<<moo; // will write the string "BlochParam: offset=0, T1=0,  
T2=0, moles=1, Spin=1H, momentum=1/2, gamma=2.6752E+8"`

## Bloch Parameters::IO

**Function:** `fstream &operator>>(const BlochParams &bp);`

**Input:** `bp-->` the BlochParams you wish to read into

**Description:** `READS` the binary data as....  
`BP{binary size}{Mo}{Moles}{Bo}{Temp}{Spin}.`

It advances the position of the file pointer.

**Example Usage:** `BlochParams moo("1H");  
fstream in(ios::binary | ios::in);  
in>>moo; // will attempt to READ the Binary Data from a file`

## Bloch Parameters::IO

**Function:** `void print(ostream &oo=cout)`

**Input:** `oo-->` an output stream

**Description:** `□` Writes the string

"BlochParam: offset={offset}, T1={T1}, T2={T2}, moles={Moles}, Spin={symbol},  
momentum={Momentum}, gamma={gamma}"  
to the ostream oo.

**Example** `□ BlochParams moo( "1H" );`

**Usage:** `moo.print() // will write the string "BlochParam: offset=0, T1=0,  
T2=0, moles=1, Spin=1H, momentum=1/2, gamma=2.6752E+8" to the  
console`

## Bloch Parameters::IO

**Function:** `void read(fstream &ii)`

**Input:** `ii-->` an INPUT BINARY stream you wish to import the data

**Description:** `READS` the binary data as....  
`BP{binary size}{Mo}{Moles}{Bo}{Temp}{Spin}`

**Example** `BlochParams moo( "1H" );`

**Usage:** `fstream in(ios::binary | ios::in);`  
`moo.read(in); // will attempt to READ the Binary of "BP {size} 0 0`  
`0 1 4.7 300 1H" to the fstream`

## Bloch Parameters::IO

**Function:** `void write(fstream &oo)`

**Input:** `oo-->` an output BINARY stream you wish to output the data

**Description:** `oo` writes the binary data as....

`BP {binarysize} {Mo} {Moles} {Bo} {Temp} {Spin}`

**Example** `BlochParams moo("1H");`

**Usage:** `fstream out(ios::binary | ios::out);`

`moo.write(out) // will write the Binary of "BP {size} 0 0 0 1 4.7  
300 1H" to the fstream`

## Bloch Parameters::other

**Function:**  **void calcMo( )**

Input:  void

Description:  Calculates the initial total magnetization of the 'spin' at equilibrium inside a large magnetic field given by the curie formula

Mo=Moles\*No\*hbar\*hbar\*gamma()\*gamma()/3./kb/Temperature\*Ho\*qn()\*(qn() + 1.);

Example  BlochParams moo( "1H" );

Usage: moo.Bo( )=4.7;

moo.calcMo( );

cout<<moo.Mo( ); //prints out something ~10E-4

## Bloch Parameters::other

**Function:**  **void setInitialCondition(InitialCondition\_t IC)**

**Input:**  IC-->an intial condition can be any 'InitialCondition'

**Description:**  IC-->an intial condition can be

It can be any of the following....

InitialCondition::AllUp --> Mo will be the max along z

InitialCondition::AllDown --> Mo will be as negative a possible along Z

InitialCondition::HalfUpHalfDown --> Will be half +Z and half -Z

InitialCondition::RandomDistribution --> Mo will be randomly distributed along the enitre bloch sphere

InitialCondition::RandomUp --> Mo will be randomly placed in the Top hemisphere

InitialCondition::RandomDown --> Mo will be randomly placed in the Bottom hemisphere

InitialCondition::RandomUpDown --> Mo will be +Z or -Z but randomly (i.e. there is a possiblility for the sample to be more up the down or vice versa)

**Example**  BlochParams moo("1H"), loo("13C");

**Usage:**  moo.setInitialCondition(InitialCondition::Random); //sets the param to a random value

**--Constructors**

--- ListBlochParams  
**--Element Extraction**  
--- Bo  
--- gamma  
--- gammaGauss  
--- meanMo  
--- Mo  
--- moles  
--- operator[], operator()  
--- Point  
--- temperature  
--- totalMo

**--Assignments**

--- operator=

**--IO**  
--- operator<<  
--- print  
--- read  
--- write

**--Other Functions**

--- +, +=  
--- calcInitialCondition  
--- calcTotalMo  
--- initialCondition  
--- setInitialCondition

**Examples**

--- 1) simple list  
--- 2) writing/saving  
--- 3) grids  
--- 4) partilces  
--- 5) Gradients

```
template<class Grid_t=NullGrid,
int BPop=BPop::Density | BPop::HighField,
class offset_T=double>
class ListBlochParams
```

---

- This class creates the interface between a 'XYZshape' or Grid and a list of Spin parameters....The 'Grid\_t' template parameter is the basic grid class that you wish to use..OR a 'NonGrid' dubbed Null Grid if there is no grid to worry about...This can also take in the 'GradientGrid' grid type whcih changes the definitions of the 'offset' function (noe the offset is position dependant).
  - **Grid\_T** --> The grid type/name, the default is NO grid (the "NullGrid")...This can be an 'XYZshape< ... >' type or 'Grid< ... >' type, If you are using gradients, then use the 'GradientGrid< ... >' type
  - **BPop**--> This is the BlochParameter Options type...See BlochParameters for the details of these options..Default is BPop::HighField | BPop::Density
  - **offset\_T**--> this specifies weather or not the offset is a 3-vector (i.e. has componenets along (x,y,z)) OR a single direction (i.e. the Z-axis)..the default is 'double' meaning the High field is only along the Z-axis...if offset\_T=coord then the High field main axis is NOT along the lab Z-axis, but along an arbitrary axis.
  - NOTE: the ListBlochParams uses ListBLochParams as its base class....
-

## ListBlochParams::constructor

### Function:

□ **ListBlochParams( )**

### Input:

□ void

### Description:

□ empty constructor--> Sets up the default Parameter for 1 BlochParam  
The Default isotope is "1H"  
The Default Bo\_=4.7 tesla  
The Default Temperature=300 K  
The Default offset, T1, T2=0;  
The Default moles=1

### Example Usage:

□ `ListBlochParam<> moo;`  
`std::string sys=moo.symbol(0); //returns "1H"`

## ListBlochParams::constructor

### Function:

□ `ListBlochParams(int numspins)`

### Input:

□ `numspins-->` the number of spins in the list

### Description:

□ Creates a list of 'numspins' protons.

### Example Usage:

□ `ListBlochParams<> moo(5);  
int len=moo.size(); //returns 5`

## ListBlochParams::constructor

- Function:** `□ ListBlochParams(int numspins, std::string spintype)`
- Input:** `□ numspins--> number of spins in the list  
spintype--> the isotope label for all the spins.`
- Description:** `□ Creates a list o BlochParams 'numspins' long all with the isotope 'spintype.'`
- Example Usage:** `□ ListBlochParams moo(10, "13C");  
cout<<moo.symbol(3); //prints out "13C"`

## ListBlochParams::constructor

**Function:** `□ ListBlochParams(int numspins, BlochParams &insert)`

**Input:** `□ numspins--> the number of spins in the list  
insert--> A BlochParams to duplicate numspins times`

**Description:** `□ creates a list of 'numspins' of the 'insert' BlochParams...`

**Example Usage:** `□ BlochParams onebp( "23Na" );  
ListBlochParams<> moo(5, onebp);  
int len=moo.size(); //returns 5  
cout<<moo.symbol(3); //prints "23Na" to the screen`

## ListBlochParams::constructor

**Function:**

- `ListBlochParams<NullGrid>(ListBlochParams<NullGrid> &dup)`
- `ListBlochParams<Grid_t>(ListBlochParams<NullGrid> &dup)`
- `ListBlochParams<Grid_t>(ListBlochParams<Grid_t> &dup)`

**Input:**

- `dup-->` the List to duplicate

**Description:**

- Creates a deep copy (different memory space) of the dup.

**Example Usage:**

- `BlochParams onebp( "23Na" );`
- `ListBlochParams<> moo(5, onebp);`
- `ListBlochParams<> Cp(moo); //the copy`
- `int len=moo.size(); //returns 5`
- `cout<<moo.symbol(3); //prints "23Na" to the screen`

## ListBlochParams::constructor

**Function:** `□ ListBlochParams(int numspins, std::string spintype, Grid_t &grid)`

**Input:** `□ numspins--> number of spins in the list`

`spintype--> the isotope label for all the spins...`

`grid--> the grid to associate with this list of parameters`

**Description:** `□ creates a list of BlochParams 'numspins' long all with the isotope 'spintype' and an associated grid`

**Example** `□ coord<> mins(-1,-1,-1), maxs(1,1,1);`

**Usage:** `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
ListBlochParams<XYZshape<XYZrect> > moo(10, "13C", myshape);
cout<<moo.symbol(3); //prints out "13C"
```

## ListBlochParams::constructor

**Function:** `□ ListBlochParams(int numspins, BlochParams &BP, Grid_t &grid)`

**Input:** `□ numspins--> number of spins in the list`

`BP--> a single BlochParams`

`grid--> the grid to associate with this list of parameters`

**Description:** `□ creates a list of BlochParams 'numspins' long all with the isotope 'spintype' and an associated grid`

**Example** `□ coord<> mins(-1,-1,-1), maxs(1,1,1);`

**Usage:** `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
BlochParams BP("23Na")
ListBlochParams<XYZshape<XYZrect> > moo(10, BP, myshape);
cout<<moo.symbol(3); //prints out "23Na"
```

## ListBlochParams::element extraction

### Function:

□ `offset_T &Bo(int i);`  
□ `offset_T &Bo(); --> FOR ITERATORS`

### Input:

□ `i--> the element you wish to extract...`

### Description:

□ Returns the Magnetic field on that spin (also can be used for assignment).

### Example Usage:

```
□ ListBlochParams<> moo(15, "27Al");
 moo.Bo(7)=5;
 moo[7].CalcMo();
 cout<<moo.Bo(7); //prints out '5'

ListBlochParams<>::iterator iter(moo);
iter.Bo()=9; //in tesla
iter.calcMo(); //recalc the MO after we change Bo
++iter;
iter.Bo()=8; //in tesla
iter.calcMo(); //recalc the MO after we change Bo
```

## ListBlochParams::element extraction

**Function:** `double gamma(int i)`

**Input:** `i-->` the element you wish to extract.

**Description:** `the gamma factor for that spin in units of Hz/Telsa`

**Example Usage:** `ListBlochParams<> moo(15, "27Al");  
cout<<moo.gamma(6); //prints out the gamma factor for Al`

## ListBlochParams::element extraction

**Function:**     □ **double gammaGauss(int i)**

**Input:**       □ i--> the element you wish to extract...

**Description:**     □ the gamma factor for that spin in units of Hz/Gauss

**Example Usage:** □ `ListBlochParams<> moo(15, "27Al");  
cout<<moo.gammaGauss(6); //prints out the gamma factor for Al`

## ListBlochParams::element extraction

**Function:**  **double meanMo()**

**Input:**  void

**Description:**  Returns the mean of the total Magnetization.

**Example Usage:** 

```
ListBlochParams<> moo(15, "27Al");
moo.Bo()=4.7;
moo.calcTotalMo();
cout<<moo.Mo(5)<<endl; //prints out the eq magnitization
cout<<moo.totalMo()<<endl;
cout<<moo.meanMo()<<endl;
```

## ListBlochParams::element extraction

### Function:

□ **offset\_T &Mo(int i);**  
□ **offset\_T &Mo(); --> FOR ITERATORS**

### Input:

□ i--> the element you wish to extract...

### Description:

□ Sets AND gets the equilibrium magnetization of the spin.

### Example Usage:

```
□ ListBlochParams<> moo(15, "27Al");
 moo.Mo(6)=1.05e-6;
 moo.calcTotalMo();
 cout<<moo.Mo(6); //prints out something ~10E-4

ListBlochParams<>::iterator iter(moo);
iter.Mo()=1.8e-9;
++iter;
iter.Mo()=1.2e-4;
```

## ListBlochParams::element extraction

**Function:**

- `double &moles(int i);`
- `double &moles(int i); --> FOR ITERATORS`

**Input:**

- `i-->` the element you wish to extract

**Description:**

- Returns the number of moles of that spin (also can be used for assignment).

**Example Usage:**

- `ListBlochParams<> moo(15, "27Al");`
- `moo.Moles(7)=0.5;`
- `moo[7].CalcMo();`
- `cout<<moo.Moles(7); //prints out '0.5'`
  
- `ListBlochParams<>::iterator iter(moo);`
- `iter.Moles()=0.1; //0.1 moles of spin 1`
- `iter.calcMo(); //recalc the MO after we change the temperature`
- `++iter;`
- `iter.Moles()=1; // 1 mole of spin 2`
- `iter.calcMo(); //recalc the MO after we change the temperature`

## ListBlochParams::element extraction

**Function:** `□ BlochParams &operator[int i]  
BlochParams &operator(int i)`

**Input:** `□ i--> the element you wish to extract.`

**Description:** `□ Returns the BlochParams At place i.`

**Example Usage:** `□ ListBlochParams<> moo(15, "27Al");  
cout<<moo[6].gammaGauss(); //prints out the gamma factor for Al  
cout<<moo(6).gamma();`

## ListBlochParams::element extraction

**Function:**  `coord<> Point() --> FOR ITERATORS`

**Input:**  none

**Description:**  Returns the current spin's coordinate.

**Example Usage:** 

```
ListBlochParams<XYZShape<XYZfull> > moo(15, "14N", myshape);
ListBlochParams<>::iterator iter(moo);
iter.Point(); //spin 1's coordinate
++iter;
iter.Point(); //spin 2's coordinate
```

## ListBlochParams::element extraction

**Function:**

- `double &temperature(int i);`
- `double &temperature(); --> FOR ITERATORS`

**Input:**

- `i-->` the element you wish to extract

**Description:**

- Returns the Temperature of the spin IN KELVIN (also can be used for assignment).

**Example Usage:**

- `ListBlochParams<> moo(15, "27Al");`
- `moo.Temperature(3)=256.0;`
- `moo[3].CalcMo();`
- `cout<<moo.Temperature(3); //prints out '256'`
  
- `ListBlochParams<>::iterator iter(moo);`
- `iter.Temperature()=400; //Temp is now 400 K for spin 1`
- `iter.calcMo(); //recalc the MO after we change the temperature`
- `++iter;`
- `iter.Temperature()=900; //Temp is now 900 K for spin 2`
- `iter.calcMo(); //recalc the MO after we change the temperature`

## ListBlochParams::element extraction

**Function:**  **double totalMo();**

**Input:**  void

**Description:**  Returns the total magnetization for the system.

**Example Usage:** 

```
ListBlochParams<> moo(15, "27Al");
moo.Bo()=4.7;
moo.calcTotalMo();
cout<<moo.Mo(5)<<endl; //prints out the eq magnitization
cout<<moo.totalMo()<<endl;
```

## ListBlochParams::assignments

**Function:** `□ ListBlochParams &operator=(const BlochParams &rhs);`

**Input:** `□ rhs--> a BlochParams`

**Description:** `□ Assigns the list to 1 BlochParam.`

**Example Usage:** `□ BlochParams moo( "27Al" );  
ListBlochParams<> koo;  
koo=moo;`

## ListBlochParams::assignments

**Function:** `□ ListBlochParams &operator=(ListBlochParams &tocopy);`

**Input:** `□ tocopy-->Another ListBlochParams`

**Description:** `□ Assigns one list from another in deep copy (separate memory space) fashion.`

**Example Usage:** `□ ListBlochParams<> moo( "27Al" );  
ListBlochParams<> koo;  
koo=moo;`

## ListBlochParams::IO

**Function:**  `ostream &operator<<(ostream &oo, ListBlochParams &bp)`

**Input:**  `bp-->` the ListBlochParams you wish to output  
 `oo-->` a output stream

**Description:**  `Writes a 'pretty' string to 'oo.'`

**Example**  `ListBlochParams<> moo(2, "1H");`

**Usage:**  `cout<<moo; // will write the string`  
 `// "Density BlochParam: moles=1, Spin=1H, momentum=1/2,`  
 `gamma=2.6752E+8"`  
 `// "Density BlochParam: moles=1, Spin=1H, momentum=1/2,`  
 `gamma=2.6752E+8"`

## ListBlochParams::IO

**Function:** `void print(ostream &oo=cout)`

**Input:** `oo-->` an output ASCII stream you wish to output the data

**Description:** `Writes an ASCII string to the ostream oo.`

**Example** `ListBlochParams moo(2, "1H");`

**Usage:** `moo.print(); // will write the string  
// "Density BlochParam: moles=1, Spin=1H, momentum=1/2,  
gamma=2.6752E+8"  
// "Density BlochParam: moles=1, Spin=1H, momentum=1/2,  
gamma=2.6752E+8"`

## ListBlochParams::IO

**Function:**  **void read(fstream &ii)**

**Input:**  ii--> an INPUT BINARY stream you wish to import the data

**Description:**  Reads the binary data as

```
LISTBLOCHPARS
BP{binary size} {Moles} {Bo} {Temp} {Spin}BP{binary size}{Moles} {Bo} {Temp}
{Spin}BP{binary size} {Moles} {Bo} {Temp} {Spin}BP{binary size} {Moles} {Bo} {Temp}
{Spin}
....BP{binary size} {Moles} {Bo} {Temp} {Spin}
ENDLISTBLOCHPARS
```

**Example**  ListBlochParams moo;

**Usage:**  fstream in(ios::binary | ios::in);
moo.read(in); // will attempt to READ the Binary Data

## ListBlochParams::IO

**Function:**  **void write(fstream &oo)**

**Input:**  oo--> an output BINARY stream you wish to output the data

**Description:**  writes the binary data as

```
LISTBLOCHPARS
BP{binary size} {Moles} {Bo} {Temp} {Spin}BP{binary size}{Moles} {Bo} {Temp}
{Spin}BP{binary size} {Moles} {Bo} {Temp} {Spin}BP{binary size} {Moles} {Bo} {Temp}
{Spin}
....BP{binary size} {Moles} {Bo} {Temp} {Spin}
ENDLISTBLOCHPARS
```

**Example**  BlochParams moo(20, "1H");

**Usage:**  fstream out(ios::binary | ios::out);
moo.write(out) // will write the Binary data

## ListBlochParams::other

**Function:**

- `ListBlochParams &operator+(const BlochParams &rhs);`
- `ListBlochParams &operator+(const ListBlochParams &rhs);`
- `ListBlochParams &operator+=(const BlochParams &rhs);`
- `ListBlochParams &operator+=(const ListBlochParams &rhs);`

**Input:**

- rhs--> another BlochParams or ListBlochParams

**Description:**

- addition or self addition of a single BlochParam, or ListBlochParams. It Adds it to the end of the list, making the list longer.

**Example**

- `BlochParams joo("1H"), loo("13C");`

**Usage:**

- `ListBlochParams<> moo(4, "14N");`

- `moo=moo+loo; //length of 5`

- `moo=moo+joo; //length of 6`

- `moo+=loo; //length of 7`

- `moo+=joo; //length of 8`

## ListBlochParams::other

**Function:**  **void calcInitialCondition();**

**Input:**  void

**Description:**  Recalculates the initial condition assuming you have used 'setInitialCondition' already. If not, the default is for the list is 'InitialCondition=InitialCondition::AllUp.'  
If the InitialCondition is of a random type, it will re-randomize the list.

**Example**  ListBlochParams<> moo(15, "27Al");

**Usage:**  //set the initial condition to random +1zhat or -1zhat  
moo.setInitialCondition(InitialCondition::RandomUpDown);  
moo.calcInitialCondition();

## ListBlochParams::other

### Function: void calcTotalMo()

Input:  void

Description:  Calculates the initial total magnetization for each spin AND calculates the TOTAL magnetization at equilibrium  
(i.e Sum\_i{ Mo[i] })

Each spin has the equilibrium magnetization given by the curie formula

$$Mo = (\gamma * hbar * \tanh(hbar * \pi * (Bo() * \gamma) / (k_b * temperature()) * moles() * N_A * 1e3 / 2.0))$$

Example  ListBlochParams<> moo(15, "27Al");

Usage:  moo.Bo()=4.7;  
moo.calcTotalMo();  
cout<<moo.Mo(5); //prints out the eq magnetization

## ListBlochParams::other

**Function:**      **int initialCondition()**

**Input:**        void

**Description:**    Returns the 'InitialCondtn' integer flag.

**Example Usage:**    `moo.setInitialCondition(InitialCondition::RandomUpDown);  
moo.calcInitialCondition();  
int IC=moo.initialCondition();`

## ListBlochParams::other

**Function:**  **void setInitialCondition(int IC);**  
 **void calcInitialCondition(int IC);**

**Input:**  IC --> the initial condition

**Description:**  Set the initial conditions flag. It can be

InitialCondition::AllUp --> Mo will be the max along z

InitialCondition::AllDown --> Mo will be as negative a possible along Z

InitialCondition::HalfUpHalfDown --> Will be half +Z and half -Z

InitialCondition::RandomDistribution --> Mo will be randomly distributed along the enitre bloch sphere

InitialCondition::RandomUp --> Mo will be randomly placed in the Top hemisphere

InitialCondition::RandomDown --> Mo will be randomly placed in the Bottom hemisphere

InitialCondition::RandomUpDown --> Mo will be +Z or -Z but randomly (i.e. there is a possiblility for the sample to be more up the down or vice versa)

It will then recaluculat the Mo of each spin. The two functions above do the same thing.

**Example**  ListBlochParams<> moo(15, "27Al");

**Usage:**  //set the initial condition to random +1zhat or -1zhat

moo.setInitialCondition(InitialCondition::RandomUpDown);

## Simple List Example

---

Demonstration on how to set up a list of Bloch Parameters.

---

```
ListBlochParams<> myPars(10); //sets up a list of 10 spins with NO grid..
cout<<myPar.gamma(0); //prints the gamma factor the first spin

ListBlochParams<NullGrid>::iterator myIt(myPars); //set up an iterator
while(myIt){
 myIt.Bo(400.0e6); //set the Bo to 400 MHz
 cout<<myIt.T1()<<" "<<myIt.Bo()<<endl; //print '400 0 400e6'
 ++myIt; //advance the iterator
}
```

---

## Saving parameters

---

How to write the parameters to a file for later reading.

---

```
BlochParams par1("13C"); //a 13C bloch parameter
BlochParams par2("1H"); //a 1H parameter

ListBLochParams<NullGrid> mypars; //an empty list
mypars+=par1; //add par1 to the list
mypars+=par2; //add par2 to the list

ListBlochParams<NullGrid> secondlist(20); //a list of 20 protons
secondlist+=mypars; //add the other list together

fstream outfile("params", ios::binary | ios::out);
secondlist.write(outfile); //write this list to a binary file for later reading...
```

---

## Using Grids with Parameters.

---

```
coord<> mins(-1,-1,-1), maxs(1,1,1);
coord<int> dims(10,10,10);

Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a rectangular grid of 1000 points

//to use the grid...the number of blochparameters should be the
// same number as those in the shape...
ListBlochParams<XYZshape<XYZrect> > mypars(myshape.size(), "1H", myshape); //a list of 1000 protons..

ListBlochParams<XYZshape<XYZrect> >::iterator myIt(mypars); //set up an iterator
while(myIt){
 myIt.Bo(400.0e6); //set the Bo to 400 MHz
 //the grid iterator is attached to this iterator also...
 // so they advance in tandem
 cout<<myIt.Point()<<endl; //print out the current grid position
 cout<<myIt.Bo()<<endl; //print the bfield at the grid position
 ++myIt; //advance the iterator
}
```

---

## Using BOptions::Particles

---

```
coord<> mins(-1,-1,-1), maxs(1,1,1);
coord<int> dims(10,10,10);

Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a rectangular grid of 1000 points

//to use the grid...the number of blochparameters should be the
// same number as those in the shape...
// typedef the name to save some typing the 'HighField' 'Particle' picture of the
// individual Bloch Spins (changes the meaing of the Magnitization)
// where the main magentic field axis is now NOT nessesarily along the z-axis...
typedef ListBlochParams<XYZshape<XYZrect>, //the grid
 BOptions::Particle | BOptions::HighField, //the Param Type option
 coord<> > //off z-axis Bfield possible
 MyList;

MyList mypars(myshape.size(), "1H", myshape); //a list of 1000 protons..

MyList::iterator myIt(mypars); //set up an iterator
while(myIt){
 myIt.Bo(coord<>(400.0e6, 200e6, 100e5)); //set the Bo to a NEW axis
 //the grid iterator is attached to this iterator also...
 // so they advance in tandem
 cout<<myIt.Point()<<endl; //print out the current grid position
 cout<<myIt.Bo()<<endl; //print the magentic field !!!a COORD<> !!!
 ++myIt; //advance the iterator
}
```

---

## Using Gradient Grids.

---

```
coord<> mins(-1,-1,-1), maxs(1,1,1);
coord<int> dims(10,10,10);

Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a rectangular grid of 1000 points

//the gradient grid
GradientGrid<TheGrid > mygrad(jj);
//to use the grid...the number of blochparameters should be the
// same number as those in the shape...
//NOTE:: placing the 'GradientGrid' class in the template definition
//automatically sets up the Gradient object a
ListBlochParams<GradientGrid< XYZshape<XYZrect> > > mypars(mygrad.size(), "1H", mygrad); //a list of 1000 protons..

ListBlochParams<GradientGrid< XYZshape<XYZrect> > ::iterator myIt(mypars); //set up an iterator
while(myIt){
 myIt.Bo(400.0e6); //set the Bo to 400 MHz
 //the grid iterator is attached to this iterator also...
 // so they advance in tandem
 cout<<myIt.Point()<<endl; //print out the current grid position

 cout<<" "<<myIt.Bo()<<" "<<myIt.Point()<<endl; //print
 ++myIt; //advance the iterator
}
```

---

**--Constructors**[--- GradientGrid](#)[--- GradientGrid](#)[--- GradientGrid](#)**--Element Extraction**[--- G](#)[--- Grid](#)[--- Gx, Gy, Gz](#)[--- offset](#)[--- Xoffset,Yoffset,Zoffset](#)**--Assignments**[--- G](#)[--- Gx, Gy, Gz](#)[--- setGrid](#)**--Other Functions**[--- operator=](#)[--- size](#)**Examples**[--- 1\) Gradients](#)

```
template<class GridEngine_t>
class GradientGrid :
public GridEngine_t
```

---

- This is the class used for in BOTH the ListBochParams' AND 'Offset<...>' classes to create a valid Gradient across a physical sample sample.
- **GridEngine\_t-->** The is the basic Grid and provides the (x,y,z) to the gradients...Everything you can do with a Grid you can do with this class (including the iterators)...however, there are extra functions for the Gradient parts..This can be either the 'XYZshape<...>' classes or the Grid<...>' classes.
- **NOTE::** this class by itself does little except act like a 'Grid' it is used primarily for the ListBlochParams and the interaction 'Offset'

**!!NOTE!!**

- **NOTE::**This is a HIGH FIELD Gradient, meaning the only part of the Gradient Tensor that matters is the Diagonal, and all the gradients contribute to an offset along the Bo direction.
  - **NOTE::** units for the Gradient strengths are assumed to be Gauss/cm
  - **NOTE::** this Gradient is only valid for a BlochParameter or ListBlochParameter with Offset\_T=double (the coord<> is in the works)
  - **NOTE::** All the functions available to GridEngine\_t are also available for this class
-

## Gradients::constructor

**Function:**     □ **GradientGrid(const GradientGrid &dup):**

Input:           □ dup--> the old GradientGrid to copy

Description:       □ Copy constructor.

Example Usage:   □ `//create a cube from (-1,-1,-1)..(1,1,1) with 10 steps`

`// on all sides`

`GradientGrid<XYZshape> > moo(-1, 1, 10);`

`GradientGrid<XYZshape> > loo(moo);`

## Gradients::constructor

**Function:** `GradientGrid(double min, double max, int dims);`

**Input:**

- `min--> min x,y,z value for the grid`
- `max--> max x,y,z value for the grid`
- `dim--> number of dimensions for the grid`

**Description:**

- `Initializes the GridEngine_t using the min, max, and dims. Sets the Gradient strength along all 3 directions to 0.`

**Example**    `//create a cube from (-1,-1,-1)..(1,1,1) with 10 steps`

**Usage:**    `// on all sides`  
`GradientGrid<XYZshape<> > moo(-1, 1, 10);`

## Gradients::constructor

**Function:**     □ **GradientGrid(GridEngine\_t &grid):**

Input:           □ grid--> a grid

Description:       □ Creates a Gradient Grid from an already initialized GridEngine.

Example Usage:   □ `//create a cube from (-1,-1,-1)..(1,1,1) with 10 steps  
// on all sides  
XYZshape<> shape(-1,1,10);  
GradientGrid<XYZshape<> > moo(shape);`

## Gradients::element extraction

### Function:

□ `coord<> &G();`

### Input:

□ `void`

### Description:

□ Returns the Gradient 3-vector. Can also be used for assignment.

### Example Usage:

```
□ XYZshape<> shape(-1,1,10);
GradientGrid<XYZshape<> > moo(shape);
moo.G()=coord<>(5,6,0); //set the Gradient
coord<> myG=moo.G(); //get the gradient
```

## Gradients::element extraction

### Function:

□ `GridEngine_t *Grid();`  
    `const GridEngine_t *Grid() const;`

### Input:

□ `void`

### Description:

□ Obtains the pointer to the GridEngine\_t inside the GradientGrid.

### Example Usage:

□ `XYZshape<> shape(-1,1,10);`  
    `GradientGrid<XYZshape<> > moo(shape);`  
    `XYZshape<> *gptr=moo.Grid();`

## Gradients::element extraction

### Function:

- `double Gx();`
- `double Gy();`
- `double Gz();`

### Input:

- `void`

### Description:

- Returns the Gradient Strength along the (x,y,z) direction in Gauss/cm.

### Example Usage:

- `XYZshape<> shape(-1,1,10);`
- `GradientGrid<XYZshape<> > moo(shape);`
- `moo.Gx(5); //set the gradient along x`
- `double gx=moo.Gx();`

## Gradients::element extraction

### Function:

```
□ double offset(int i);
double offset(int i, double t);

double offset(); --> FOR ITERATORS
double offset(double t); --> FOR ITERATORS
```

### Input:

□ i--> the grid point index

t--> a time

### Description:

□ Returns the total offsets along each three directions using

offset=r(x,y,z) .dot. G(x,y,z).

If r=r(t) (the grid is time dependant) use the overloaded function for include time.

### Example Usage:

```
□ XYZshape<> shape(-1,1,10);
GradientGrid<XYZshape<> > moo(shape);
double offs=moo.offset(5);
GradientGrid<XYZshape<> >::iterator iter(moo);
offs=iter.offset();
```

## Gradients::element extraction

**Function:**

- `double Xoffset(int i);`
- `double Yoffset(int i);`
- `double Zoffset(int i);`
  
- `double Xoffset(int i, double t);`
- `double Yoffset(int i, double t);`
- `double Zoffset(int i, double t);`
  
- `double Xoffset(); --> FOR ITERATORS`
- `double Yoffset(); --> FOR ITERATORS`
- `double Zoffset(); --> FOR ITERATORS`
  
- `double Xoffset(double t); --> FOR ITERATORS`
- `double Yoffset(double t); --> FOR ITERATORS`
- `double Zoffset(double t); --> FOR ITERATORS`

**Input:**

- `i--> the grid point index`
- `t--> a time`

**Description:**

- Returns the (x,y,z) offset FROM THE GRADIENT (i.e.  $x^*G_x, y^*G_y, z^*G_z$ ) not from a spins offset.  
If the GridEngine\_t is time dependant (i.e.  $r=r(t)$ ) then use the overload function.

**Example**

- `XYZshape<> shape(-1,1,10);`

**Usage:**

- `GradientGrid<XYZshape<> > moo(shape);`
- `moo.Xoffset(5);`
- `GradientGrid<XYZshape<> >::iterator iter(moo);`
- `iter.Xoffset();`

## Gradients::assignments

### Function:

```
□ void G(coord<> &inG);
 void G(double Gx, double Gy, double Gz);
```

### Input:

```
□ inG--> a 3-vector coord
 Gx, Gy, Gz--> the gradient along all three directions
```

### Description:

```
□ Set the Gradient 3 vector in Gauss/cm.
```

### Example Usage:

```
□ XYZshape<> shape(-1,1,10);
GradientGrid<XYZshape<> > moo(shape);
moo.G(coord<>(5,5,5));
moo.G(5,5,5); //the same thing as above
```

## Gradients::assignments

### Function:

- `void Gx(double inG);`
- `void Gy(double inG);`
- `void Gz(double inG);`

### Input:

- `inG--> a double in Guass/cm`

### Description:

- Assigns the Gradient along all three directions.

### Example Usage:

- `XYZshape<> shape(-1,1,10);`
- `GradientGrid<XYZshape<> > moo(shape);`
- `moo.Gx(5); //set the gradient along x`
- `double gx=moo.Gx();`

## Gradients::assignments

### Function:

□ **void setGrid(GridEngine\_t &in);**

### Input:

□ in--> a grid

### Description:

□ Set the GridEngine\_t to a new one.

### Example Usage:

```
□ XYZshape<> shape(-1,1,10);
 GradientGrid<XYZshape<> > moo;
 moo.setGrid(shape);
```

## Gradients::other

**Function:** `void operator=(const GradientGrid &rhs)`

**Input:** `rhs-->` another GradientGrid

**Description:** `Assigns a GradientGrid from another.`

**Example Usage:**

```
XYZshape<> shape(-1,1,10);
GradientGrid<XYZshape<> > moo(shape);
GradientGrid<XYZshape<> > loo;
loo=moo;
```

## Gradients::other

### Function:

**int size()**

### Input:

void

### Description:

Obtains the size of the grid.

### Example Usage:

```
 XYZshape<> shape(-1,1,10);
 GradientGrid<XYZshape<> > moo(shape);
 int ss=moo.size();
```

## Using Gradients with ListBlochParams

---

This example demonstrates how to create a GradientGrid using the XYZshapes as the base. It then uses the GradientGrid as the Grid for the 'ListBlochParams.' The ListBlochParams is then assigned as a pointer to the BlochInteraction 'Offset' where the Gradient offsets are calculated on top of a global spin offset.

---

```
//create a 10x10x10 grid
coord<> mins(-1,-1,-1), maxs(1,1,1);
coord<int> dims(10,10,10);
Grid<UniformGrid> grid(mins, maxs, dims);

//a full cylinder of radius 0.5 and length 2
typedef XYZshape<XYZcylinder> TheShape;
coord<> smins(0,0,-1), smaxs(0.5, PI2, 1);
XYZcylinder inshape(smins, smaxs);
TheShape myshape(grid, inshape);

//set up the a gradient grid
typedef GradientGrid<TheShape> TheGrid;
TheGrid gradgrid(myshape);

//set up the gradient parameters
// for Gx=0, Gy=0, Gz=5 Gauss/cm
gradgrid.G(0.0,0.0, 5.0);

//setup the list of bloch parameters
//using the gradient grid as its base
typedef ListBlochParams< TheGrid> MyPars;
MyPars mypars(gradgrid.size(), "1H", gradgrid);

//now set up the Offset Interaction using the gradeint grid list of parameters
double offset=100.0*PI2; //a 100 Hz offset frequency for all the spins
Offset<MyPars> myOffs(mypars, offset);
myOffs.on(); //turn on the gradient in the offsets

//USING THE ITERATOR
MyPars::iterator myit(mypars);
while(myit){
 cout<<myOffs.offset(myit.curpos())<<endl; //the will print out dot(G, Point())
 ++myit;
}
```

---

**--Constructors**

--- RotatingGrid

--- RotatingGrid

--- RotatingGrid

--- RotatingGrid

**--Element Extraction**

--- axis

--- operator(i,t)

--- Point

--- rate

**--Assignments**

--- operator=

--- setAxis, axis

--- setRate, rate

**Examples**

--- 1) rotating grid

```
template<class ShapeEng_t>
class RotatingGrid :
public ShapeEng_t
```

---

Like the Gradient Grid, this is an extention to the Grid class, to allow for rotation of grids about one axis.

---

## RotatingGrid::constructor

### Function:

- **RotatingGrid()**

### Input:

- void

### Description:

- Default constructor. Set the default values
  - rate=0
  - axis=(0,0,1)

### Example Usage:

- `RotatingGrid<XYZshape<> > grid;`

## RotatingGrid::constructor

**Function:**  **RotatingGrid(const RotatingGrid &cp);**

**Input:**  cp--> a RotatingGrid to copy

**Description:**  The copy constructor copies one Rotating Grid to the next. The ShapeEng\_t must be the same for BOTH the copy and the copier.

**Example**  `RotatingGrid<XYZshape<> grid;`

**Usage:**  `RotatingGrid<XYZshape<> grid2(grid);`

## RotatingGrid::constructor

**Function:**  `RotatingGrid(const ShapeEng_t &mygrid, const coord<> &axis=coord<>(0,0,0), const double &rate=0.0, bool continous=false ):`

**Input:**  mygrid--> an XYZshape of some kind

axis--> the axis of grid rotation

rate--> the spinning rate

continous--> is the grid time dependant or is this a static rotation?

**Description:**  The main constructor. Sets the grid to the mygrid, the axis of rotation to 'axis' and the spinning rate to 'rate.' It also sets the 'continous' flag which if set to FALSE the grid will be rotated to the 'axis' but NOT be time dependant (i.e. not spinning); if it is set to TRUE then the grid assumes the grid points spin in time.

**Example**  `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`

`XYZfull tester; //the basic shape`

`XYZshape<XYZfull> jja(gg, tester); //the XYZshape`

`RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The Rotating grid`

**Usage:**

## RotatingGrid::constructor

**Function:** □ `RotatingGrid(const coord<> &mins, const coord<> &maxs, const coord<int> &dims, const coord<> &axis=coord<>(0,0,0), const double &rate=0.0, bool continous=false );`

**Input:** □ mins--> the minima for the ShapeEng\_t  
maxs--> the maxima for the ShapeEng\_t  
dims--> the dimensions for ShapeEng\_t  
axis--> the axis of grid rotation  
rate--> the spinning rate  
continous--> is the grid time dependant or is this a static rotation?

**Description:** □ A main constructor, that allows you to set the grid parameters on the fly. Sets the axis of rotation to 'axis' and the spinning rate to 'rate.' It also sets the 'continous' flag which if set to FALSE the grid will be rotated to the 'axis' but NOT be time dependant (i.e. not spinning); if it is set to TRUE then the grid assumes the grid points spin in time. For use primarily with ShapeEng\_t='XYZshape<XYZfull>'

**Example** □ `RotatingGrid<XYZshape<XYZfull> > jj(mins, maxs, dims, myAxis, rate); //The Rotating grid`  
**Usage:**

## RotatingGrid::element extraction

**Function:** `coord<> axis();`

**Input:** `void`

**Description:** `Retrieve the axis of rotation.`

**Example** `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`

**Usage:** `XYZfull tester; //the basic shape`

`XYZshape<XYZfull> jja(gg, tester); //the XYZshape`

`RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The Rotating grid`

`jj.axis(coord<>(1,1,1));`

`coord<> ax=jj.axis(); //will be [1/sqrt(3), 1/sqrt(3), 1/sqrt(3)]`

## RotatingGrid::element extraction

**Function:**  `coord<> operator()(int i, double t);`  
`coord<> operator()(double t); --> FOR ITERATORS`

**Input:**  `i-->` the grid point index  
`t-->` a time

**Description:**  Retrieves the grid point at index `i` as a function of time `t`.  
if 'continous' is false, this will return the rotated grid obtained upon construction.

**Example**  `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`  
**Usage:** `XYZfull tester; //the basic shape`  
`XYZshape<XYZfull> jja(gg, tester); //the XYZshape`  
`RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The`  
`Rotating grid`  
`jj.axis(coord<>(1,1,1));`  
`coord<> myPt=jj(4, 0.0); //get the grid point 4 at t=0`

## RotatingGrid::element extraction

**Function:**  `coord<> Point(int i, double t);`  
`coord<> Point( double t); --> FOR ITERATORS`

**Input:**  `i--> the grid point index`  
`t--> a time`

**Description:**  Retrieves the grid point at index i as a function of time t.  
if 'continous' is false, this will return the rotated grid obtained upon construction.

**Example**  `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`  
**Usage:**  `XYZfull tester; //the basic shape`  
`XYZshape<XYZfull> jja(gg, tester); //the XYZshape`  
`RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The`  
`Rotating grid`  
`jj.axis(coord<>(1,1,1));`  
`coord<> myPt=jj.Point(4, 0.0); //get the grid point 4 at t=0`

## RotatingGrid::element extraction

**Function:** `double rate()`

**Input:** `void`

**Description:** `Returns the spinning rate.`

**Example** `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`  
**Usage:** `XYZfull tester; //the basic shape`  
`XYZshape<XYZfull> jja(gg, tester); //the XYZshape`  
`RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The`  
`Rotating grid`  
`jj.rate(5600*PI2);`  
`double rt=jj.rate(); //5600*PI2`

## RotatingGrid::assignments

**Function:** `void operator=(const RotatingGrid &rhs)`

**Input:** `rhs-->` another RotatingGrid

**Description:** `Assignment operator.` The ShapeEng\_t for both the opy and the copier must be the same.

**Example** `Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid`

**Usage:** `XYZfull tester; //the basic shape`

```
XYZshape<XYZfull> jja(gg, tester); //the XYZshape
RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The
Rotating grid
RotatingGrid<XYZshape<XYZfull> > jj2;
jj2=jj;
```

## RotatingGrid::assignments

**Function:**  **void setAxis(const coord<> &ax);**  
 **void axis(const coord<> &ax);**

**Input:**  ax--> a valid axis.

**Description:**  Sets the rotation axis. The axis vector will be Normalized.

**Example**  Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid  
**Usage:**  XYZfull tester; //the basic shape  
XYZshape<XYZfull> jja(gg, tester); //the XYZshape  
RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The  
Rotating grid  
jj.axis(coord<>(1,1,1));

## RotatingGrid::assignments

**Function:**  **void setRate(const double &rt);**  
**void rate(const double &rt)**

**Input:**  rt--> a rate in Rads/Sec

**Description:**  Sets the Spinning rate. This should be in RADIANS/s.

**Example**  Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid

**Usage:**  XYZfull tester; //the basic shape  
XYZshape<XYZfull> jja(gg, tester); //the XYZshape  
RotatingGrid<XYZshape<XYZfull> > jj(jja, myAxis, rate); //The  
Rotating grid  
jj.rate(500.0\*PI2);

## Rotating Grid example

---

Simpley sets up a Rotating Grid. It behaves exactly as the static grids do except that the grid points are now a function of time. This class is meant to emulate a MAS type experiment.

---

```
//some typedefs to make typeing easier
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrida; //our shape
typedef RotatingGrid<TheGrida> TheGrid; //spinning grid

//this is our ListBlochParams typedef
typedef ListBlochParams<
 TheGrid,
 BPOptions::Particle | BPOptions::HighField,
 double > MyPars;

coord<> myAxis(1,1,1); //the magic angle axis
double rate=2000.0*PI2; //the spinning rate in radians

Grid<UniformGrid> gg(mins, maxs, dims); //the basic grid

TheShape tester; //the basic shape
TheGrida jja(gg, tester); //the XYZshape
TheGrid jj(jja, myAxis, rate); //The Rotating grid

//an initial condition
int IC=InitialCondition::Random;
//create a parameter set
//with the same number of spins as
// grid points, all protons.
// with our rotating grids
// and a random initialcondition
MyPars mypars(jj.size(), "1H", jj, IC);
//set some properties of the params
for(int j=0;j<mypars.size();j++){
 mypars.temperature(300.0);
}

//recalc the total Mo
mypsars.calcTotalMo();

//use the parameters as your wish.....
```

---

**--Public Vars**

--- SPulse::End

--- SPulse::Start

**--Constructors**

--- Pulse

--- Pulse

--- Pulse

**--Element Extraction**

--- amplitude

--- beginTime

--- coordPulse

--- endTime

--- offset

--- operator[i]

--- phase

--- symbol

--- Xpart, Ypart, Zpart

**--Assignments**

--- operator=

--- operator=

--- setAmplitude

--- setBeginTime

--- setEndTime

--- setOffset

--- setPhase

--- setSymbol

--- setTime

**--IO**

--- operator<<

--- print

**--Other Functions**

--- +, +=

--- size

--- timeForAngle

**Examples**

--- 1) simple setup

```
class Pulse{
```

```
...
```

- 
- Like GradientGrid, this class acts as a parameter container for other classes (namely the class 'bloch'). It stores the list of amplitudes, phases, offsets, and pulse times, and spin types, and calculates the desired B1 (pulse field strength) at for a give time

!!NOTE!!

- **NOTE:** there is a 'data-element' class called 'SPulse' can contains individual spin's pulse parameters...below in the function descriptions for 'Pulse' you will find the compatable functions with 'SPulse'
- **NOTE:** all units should be entered as Radains/s NOT Hz (i.e. multiply everything by a 2Pi..a 40kHz B1 field would have an inputted amplitude of 40000\*2\*pi)

## Pulses::Public Vars

### Function: SPulse::End

Input:

Description:  A time flag to tell the class that the time domain for the pulse is anywhere AFTER the start time.  
Using <SPulse::Start...SPulse::End>, means ALL time is available to the pulse.

Example

Usage:

## Pulses::Public Vars

**Function:**  **SPulse::Start**

**Input:**

**Description:**  Time flag to tell the class that the time domain for the pulse is anywhere BEFORE the 'end.'

**Example Usage:**

## Pulses::constructor

**Function:**  **Pulse()**

**Input:**  void

**Description:**  empty constructor--> Sets up the default Pulse  
The Default isotope is "1H"  
The Default amplitude=0.0 rad/s  
The Default phase=0.0  
The Default offset=0.0  
The Default beginTime=SPulse::Start  
The Default endTime=SPulse::End

**Example**

**Usage:**  Pulse moo; //a pulse that is on 1H for any time (but no amplitude)

## Pulses::constructor

**Function:**  `Pulse(std::string symbol, double amp=0.0, double phase=0.0, double offset=0.0, double tbegin=SPulse::Start, double tend=SPulse::End)`

**Input:**  symbol--> the name of the atom...in the format "<atomic mass><periodic label>" like "1H", "2H", "7Li"...etc  
amp--> the input amplitude (default will be 0)  
phase-->the input pulse phase IN RADIANS (default 0.0)  
offset-->input offset (default is 0.0)  
tbegin-->begin time for the pulse (default is SPulse::Start)  
tend-->end time for the pulse (default is SPulse::End)

**Description:**  Creates a SINGLE entry in the pulse list for the nucleus with symbol.

**Example**  `//a 40 kHz pulse on proton`

**Usage:**  `// to be applied at all times`

`Pulse moo( "1H", 40000.0*PI2 );`

## Pulses::constructor

### Function:

□ **Pulse(const Pulse &cp)**

### Input:

□ cp--> an existing Pulse to copy..

### Description:

□ Copy constructor.

### Example Usage:

```
□ // a 40 kHz pulse on proton
// to be applied at all times
Pulse moo("1H", 40000.0*PI2);
Pulse loo(moo);
```

## Pulses::element extraction

- Function:**
- `double amplitude(int i)`
  - `double amplitude(std::string sym)`
  - `double amplitude(double time, int i)`
  - `double amplitude(double time, std::string sym)`
- Input:**
- `i`--> the element number in the list
  - `sym`-->will find the amplitude for this atomic symbol (i.e. '1H')
  - `time`-->will find the amplitude for the given index or symbol if time is between begining and end times
- Description:**
- Returns the amplitude inside the list, will return 0 if not found.
- Example**
- `Pulse moo( "13C", 30000, PI/2. );`
  - `cout<<moo.amplitude(0); //returns 30000`
  - `cout<<moo.amplitude( "13C"); //returns 30000`
  - `cout<<moo.amplitude( "1H"); //returns 0`
  - `cout<<moo.amplitude(1e-3, "13C"); //returns 30000 (as all time is`
- Usage:**

## Pulses::element extraction

### Function:

- `double beginTime(int i)`
- `double beginTime(std::string sym)`

### Input:

- `i`--> the element number in the list
- `sym`-->will find the begin time for this atomic symbol (i.e. '1H')

### Description:

- Returns the beginning time for a symbol of the element in the list.

### Example Usage:

- `Pulse moo("13C", 30000, PI/2., 50.0, 0.001, 0.002);`
- `cout<<moo.endTime("13C"); //returns 0.002`
- `cout<<moo.endTime("1H"); //returns 0.0 (not present)`

## Pulses::element extraction

**Function:** `coord<> coordPulse(double time, std::string sym)`

**Input:** `sym`-->will find the magnetic field for this atomic symbol (i.e. '1H')  
`time`-->will find the magnetic field for the given index or symbol if time is between begining and end times

**Description:** `>Returns the 3-vector of the Magnetic B1 field if there is any pulse specified in the time 'time.'`

`x=amplitude*sin(phase)`  
`y=amplitude*cos(phase)`  
`z=offset`

**Example** `Pulse moo("13C", 30000, PI/2., 50.0);`

**Usage:** `cout << moo.CoordPulse(0, "13C"); //returns [ 30000*sin(PI/2) 30000*cos(PI/2) 50 ]`

## Pulses::element extraction

### Function:

□ `double endTime(int i)`  
□ `double endTime(std::string sym)`

### Input:

□ `i`--> the element number in the list  
`sym`-->will find the end time for this atomic symbol (i.e. '1H')

### Description:

□ Returns the end time for a symbol of the element in the list.

### Example Usage:

```
□ Pulse moo("13C", 30000, PI/2., 50.0, 0.001, 0.002);
cout<<moo.endTime("13C"); //returns 0.002
cout<<moo.endTime("1H"); //returns 0.0 (not present)
```

## Pulses::element extraction

**Function:**

- `double offset(int i)`
- `double offset(std::string sym)`
- `double offset(double time, int i)`
- `double offset(double time, std::string sym)`

**Input:**

- `i`--> the element number in the list
- `sym`-->will find the offset for this atomic symbol (i.e. '1H')
- `time`-->will find the offset for the given index or symbol if time is between begining and end times

**Description:**

- Returns the pulse offset inside the list, will returns 0 if not found.

**Example**

- `Pulse moo("13C", 30000, PI/2., 50.0);`

**Usage:**

- `cout << moo.offset(0); //returns 50`
- `cout << moo.offset("13C"); //returns 50`
- `cout << moo.offset("1H"); //returns 0`
- `cout << moo.offset(1e-3, "13C"); //returns 50 (as all time is allowed)`

## Pulses::element extraction

**Function:** `SPulse &operator[int i],  
Spulse &operator(int i)`

**Input:** `i-->` the element number in the list

**Description:** `Returns the single Pulse element in the list .`

**Example Usage:** `Pulse moo("13C", 30000, PI/2., 50.0, 0.001, 0.002);  
cout<<moo[0].endTime(); //returns 0.002  
cout<<moo[1].endTime(); //ACKK will crash the machine...`

## Pulses::element extraction

**Function:**

- `double phase(int i)`
- `double phase(std::string sym)`
- `double phase(double time, int i)`
- `double phase(double time, std::string sym)`

**Input:**

- `i`--> the element number in the list
- `sym`-->will find the phase for this atomic symbol (i.e. '1H')
- `time`-->will find the phase for the given index or symbol if time is between begining and end times

**Description:**

- Returns the phase (in radians) inside the list, will return 0 if not found.

**Example**

- `Pulse moo("13C", 30000, PI/2.);`

**Usage:**

- `cout<<moo.phase(0); //returns Pi/2`
- `cout<<moo.phase("13C"); //returns Pi/2`
- `cout<<moo.phase("1H"); //returns 0`
- `cout<<moo.phase(1e-3, "13C"); //returns Pi/2 (as all time is allowed)`

## Pulses::element extraction

**Function:** `std::string symbol(int i)`

**Input:** `i-->` the element number in the list

**Description:** `Returns the symbol name of the ith element in the list...will return a NULL string ("") if there is no i'th element.`

**Example** `Pulse moo("13C", 30000*PI2, PI/2. );`

**Usage:** `std::string sys=moo.symbol(0); //returns "13C"`  
`sys=moo.symbol(1); //returns "" (a NULL string)`

## Pulses::element extraction

**Function:**  `double Xpart(double time, std::string sym),`  
`double Ypart(double time, std::string sym),`  
`double Zpart(double time, std::string sym)`

**Input:**  `sym`-->will find the magnetic field for this atomic symbol (i.e. '1H')  
`time`-->will find the magnetic field for the given index or symbol if time is between begining and end times

**Description:**  Returns the Magnetic B1 field along the respective axis. If there is any pulse specified in the time 'time'

`Xpart=amplitude*sin(phase)`

`Ypart=amplitude*cos(phase)`

`Zpart=offset`

**Example**  `Pulse moo( "13C" , 30000 , PI/2., 50.0);`

**Usage:** `cout<<moo.Xpart(0, "13C"); //returns 30000*sin(PI/2)`

`cout<<moo.Ypart(4,"1H"); //returns 0 (no 1H preset`

## Pulses::assignments

**Function:** `Pulse &operator=(const Pulse &rhs);`

**Input:** `rhs-->` the item you wish to copy

**Description:** `Assigns one Pulse from another.`

**Example Usage:**

```
Pulse moo("27Al" , 30000);
Pulse koo;
koo=moo;
```

## Pulses::assignments

**Function:**  **Pulse &operator=(const SPulse & singlepulse);**

**Input:**  singlepulse--> A SPulse data element

**Description:**  Assigns a pullse list to the SINGLE pulse. It will make the list size 1.

**Example Usage:**  Pulse moo=SPulse("1H", 40000\*PI2, PI);

## Pulses::assignments

**Function:**  `void setAmplitude(double newin, int i)`  
 `void setAmplitude(double newin, std::string sym)`

**Input:**  `newin`-->the new value  
`i`-->the element in the list  
`sym`--> a symbol name (i.e. '1H', '13C' ...)

**Description:**  Sets the amplitude of the `i`'th element in the list or the element with the symbol '`sym`'.

**Example Usage:**  `Pulse koo;`  
`koo.setAmplitude( 50000,0);`  
`koo[0].setAmplitude(50000);`

## Pulses::assignments

**Function:**  `void setBeginTime(double newin, int i)`  
`void setBeginTime(double newin, std::string sym)`

**Input:**  `newin`-->the new value  
`i`-->the element in the list  
`sym`--> a symbol name (i.e. '1H', '13C' ...)

**Description:**  Sets the begining time of the i'th element in the list or the element with the symbol 'sym.'

**Example Usage:** 

```
Pulse koo;
koo.setBeginTime(0.0, 0);
koo[0].setBeginTime(0.0);
```

## Pulses::assignments

**Function:**  `void setEndTime(double newin, int i)`  
 `void setEndTime(double newin, std::string sym)`

**Input:**  `newin`-->the new value  
`i`-->the element in the list  
`sym`--> a symbol name (i.e. '1H', '13C' ...)

**Description:**  Sets the ending time of the `i`'th element in the list or the element with the symbol '`sym`'.

**Example Usage:**  `Pulse koo;`  
`koo.setEndTime(1.0, 0);`  
`koo[0].setEndTime(1.0);`

## Pulses::assignments

- Function:**
- `void setOffset(double newin, int i)`
  - `void setOffset(double newin, std::string sym)`
- Input:**
- `newin`-->the new value
  - `i`-->the element in the list
  - `sym`--> a symbol name (i.e. '1H', '13C' ...)
- Description:**
- Sets the offset of the `i`'th element in the list or the element with the symbol '`sym`'.
- Example Usage:**
- `Pulse koo;`
  - `koo.setOffset( 400.0, 0 );`
  - `koo[0].setOffset(400.0);`

## Pulses::assignments

- Function:**
- `void setPhase(double newin, int i)`
  - `void setPhase(double newin, std::string sym)`
- Input:**
- `newin`-->the new value
  - `i`-->the element in the list
  - `sym`--> a symbol name (i.e. '1H', '13C' ...)
- Description:**
- Sets the phase of the `i`'th element in the list or the element with the symbol '`sym`'.
- Example Usage:**
- `Pulse koo;`
  - `koo.setPhase( PI/2, 0);`
  - `koo[0].setPhase(PI/2);`

## Pulses::assignments

### Function:

□ **void setSymbol(std::string sym,int i)**

### Input:

- i-->the element in the list
- sym--> a symbol name (i.e. '1H', '13C' ...)

### Description:

- Sets the symbol of the i'th element in the list.

### Example Usage:

□ Pulse koo;  
koo.setSymbol(0, "1H");  
koo[0].setSymbol("1H");

## Pulses::assignments

**Function:**  **void setTime(double begin,double end,int i)**  
 **void setTime(double begin, double end, std::string sym)**

**Input:**  begin-->the new begin value  
 end-->the new end value  
 i-->the element in the list  
 sym--> a symbol name (i.e. '1H', '13C' ...)

**Description:**  Sets the begining and ending time of the i'th element in the list or the element with the symbol 'sym.'

**Example**  Pulse koo;

**Usage:**  koo.setTime(0.0, 1.0, "1H");  
 koo[0].setTime(0.0, 1.0);

## Pulses::IO

**Function:** `ostream &operator<<(ostream &oo, Pulse &out)`

**Input:** `oo-->` an output ASCII stream you wish to output the data  
`out-->` the Pulse you wish to output

**Description:** `W` Writes the string....

"Pulse on {symbol} B1: {amplitude} phase: {phase} offset: {offset} begin {btime} end: {endt}"  
to the ostream.

**Example** `Pulse PP1("1H", 80000, 0.); // (spin, amplitude, phase)`

**Usage:** `cout<<PP1<<endl; //prints "Pulse on 1H B1: 80000 phase: 0" to the console`

## Pulses::IO

**Function:**  **void print(ostream &oo=cout)**

**Input:**  oo--> an output ASCII stream you wish to output the data

**Description:**  Writes the string....

"Pulse on {symbol} B1: {amplitude} phase: {phase} offset: {offset} begin {btime} end: {endt}"  
to the ostream.

**Example Usage:**  Pulse PP1("1H", 80000, 0.); // (spin, amplitude, phase)  
//prints "Pulse on 1H B1: 80000 phase: 0" to the console  
PP1.print(cout);

## Pulses::other

### Function:

- `Pulse operator+(const Pulse &rhs);`
- `Pulse operator+=(const Pulse &rhs);`
  
- `Pulse operator+(const SPulse &rhs);`
- `Pulse operator+=(const SPulse &rhs);`

### Input:

- `rhs--> another Pulse`

### Description:

- Adds an existing Pulse or SPulse to the current Pulse

### Example Usage:

- `Pulse moo("1H", 20000), loo("13C", 80000);`
- `loo+=moo; //both 1H and 13C are present`
- `Pulse koo=loo+moo; //both 1H and 13C are present`
- `loo=moo+SPulse("13C", 40000, Pi/2);`
- `moo+=SPulse("13C", 40000, Pi/2);`

## Pulses::other

**Function:**  **int size()**

**Input:**  void

**Description:**  Returns the number of Pulses in the Pulse list.

**Example Usage:**  Pulse PP1( "1H", 80000, 0.); // (spin, amplitude, phase)  
PP1.size(); //length=1  
PP1+=SPulse("13C", 50000, 0.);  
PP1.size(); //length=2

## Pulses::other

**Function:** `double timeForAngle(double angle, string symbol)`

**Input:** `angle--> the desired angle you wish to pulse IN RADIANS!!`  
`symbol--> the spin symbol`

**Description:** `>Returns the time for a given pulse angle (in radians) for the spin 'symbol' calculated by angle/amplitude.`  
`Returns 1e-30 if spin Not found or angle/amplitude < 0`

**Example** `Pulse PP1("1H", 80000, 0.); // (spin, amplitude, phase)`

**Usage:** `//figure out how long a '90 degree pulse' will take for the 1H`  
`double tpulse=PP1.timeForAngle(90.0*Pi/180., "1H");`

## Setting up a Pulse Interaction

---

This example shows you how to set up a Pulse interaction.

---

```
Pulse PP1("1H", 80000, 0.); // (spin, amplitude, phase)
PP1.print(cout); //prints "Pulse on 1H B1: 80000 phase: 0" to the console
//figure out how long a '90 degree pulse' will take for the 1H
double tpulse=PP1.timeForAngle(90.0*Pi/180., "1H");

//add another nucleus to pulse, but with a different phase
//NOTE: because no 'start time' and 'end time' have been entered, both
// 13C and 1H will be applied when ever this pulse is called
//even if the amplitudes are different
PP1+=SPulse("13C", 100000, PI/2.0, 500.0*PI2); //(spintype, amplitude, phase, offset)
```

---

**--Public Vars****--- amps****--- Coil****--- nloops****--Constructors****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--- BiotCoil****--Element Extraction****--- constField****--- type****--Assignments****--- constField****--IO****--- print****--- read****--- read****--- read****--- read****--- readShape****--- writeShape****--Other Functions****--- min,max****--- size****--- translate**

## class BiotCoil

- This class is simply a container to hold and generate Coils for use in the Magnetic Field calculator. The coils can be represented as either a 'function' (i.e. we know the form for a circular coil) or as an input file. The coils are defined in Cartesian space.
- It is heavily parameter based, meaning it takes a Parameters object to extract all of the relevant parameters. There are several built-in functional shapes. Below you will find the relevant shape, and the input required for that shape to function from a Parameter object.
- ----- CONSTANT FIELDS-----
  - # the type is a constant field  
type constant  
# this is a Magnetic Field Strength in Gauss!  
# along each axis {x,y,z}  
constField 1.0,1.0,1.0
- ----- FILES-----
  - #the type is a file  
type file  
  
#the file name of the shape  
# the file should either be of the form  
# -----  
# xpt1 ypt1 zpt1  
# xpt2 ypt2 zpt2  
# ....  
# -----  
# the file can have an 'amps' and a  
'loops' specified like so  
#-----  
# amps 4  
# loops 5  
# xpt1 ypt1 zpt1  
# xpt2 ypt2 zpt2  
# ....  
# if no amps or loops are define they  
default to 1 each  
#  
# the file can also be a saved file from  
a previous  
# BiotCoil (or Biot, or MultiBiot) class  
# such that multiply shapes can be saved  
and read  
  
filename shape1.biot
- ----- THE LINE---

- #the type of shape  
type line  
# the number of wraps for the line  
loops 1000  
#the current through the shape (in Amps)  
amps 2  
# number of points to chop up the shape  
for calculations  
numpts 4000  
#the {x,y,z} beginning of the line in cm  
begin 10, -150, -10  
#the {x,y,z} End of the line in cm  
end 10, 150, -10  
#the translate value...it will move every  
point  
# by the value below  
center 0,0,0

- ----- THE CIRCLE---
  - #the type of shape  
type circle  
  
# the number of wraps for the line  
loops 10  
  
#the current through the shape (in Amps)  
amps 2  
  
# number of points to chop up the shape  
for calculations  
numpts 5000  
  
#the radius (in cm) of the circle  
R 5  
  
#the translate value...it will move every  
point  
# by the value below  
center 0,0,0  
  
#the axis about which the coil will be  
generated AROUND  
# either 'x', 'y', or 'z'  
axis x
- ----- THE SPIRAL---
  - #the type of shape  
# will create a spiral in a around the  
'axis'  
type spiral  
  
# the number of wraps for the line

```

loops 10

#the current through the shape (in Amps)
amps 2

number of points to chop up the shape
for calculations
numpts 5000

#the Outer Radius (the starting radius)
(in cm) of the spiral
R1 5

#the Inner Radius (the ending radius)
(in cm) of the spiral
R2 5

#the number of turns between R1 and R2
turns 10

#the translate value...it will move every
point
by the value below
center 0,0,0

#the axis about which the coil will be
generated AROUND
either 'x', 'y', or 'z'
axis x

• ----- THE HELIX---
○ #the type of shape
type helix

the number of wraps for the line
loops 10

#the current through the shape (in Amps)
amps 2

number of points to chop up the shape
for calculations
numpts 5000

#the radius (in cm) of the helix
R 5

#the length of the helix in cm
if no 'center' is specified it will
move from
-1.5 to +1.5 cm about the 'axis'
defined below
Z 3

```

```

#the number of turns of the helix
there will be 5 turns from -1.5 to 1.5
turns 5

#the translate value...it will move every
point
by the value below
center 0,0,0

#the axis about which the coil will be
generated AROUND
either 'x', 'y', or 'z'
axis z

• ----- THE HELMHOLTZ---
○ #the type of shape
generates an IDEAL (no width, no
height)
helmholtz pair of circles
type helmholtz

the number of wraps for the line
loops 10

#the current through the shape (in Amps)
amps 2

number of points to chop up the shape
for calculations
numpts 5000

#the radius (in cm) of the helix
R 5

#the length (in cm) the two circles are
apart
if no 'center' is specified it one
circle will be at
-1.5 the other will be at +1.5 about
the 'axis'
length 3

#the translate value...it will move every
point
by the value below
center 0,0,0

#the axis about which the coil will be
generated AROUND
either 'x', 'y', or 'z'
axis z

```

- ----- THE TRUE HELMHOLTZ---

- #the type of shape
 

```
generates an REAL helmholtz pair of
spirals having depth, width, height,
and wraps
type truehelmholtz
```
- # the number of wraps for the helmholtz
 

```
this should be 1, as the number of
wraps
will be calculated as real points
loops 1
```
- #the current through the shape (in Amps)
 

```
amps 2
```
- # number of points to chop up the shape
 

```
for calculations
numpts 5000
```
- #the radius (in cm) of the helix
 

```
R 5
```
- #the length (in cm) the two circles are
 

```
apart
if no 'center' is specified it one
circle will be at
-1.5 the other will be at +1.5 about
the 'axis'
length 3
```
- #width between 2 helix layers (in cm)
 

```
layerwidth 0.161
```
- #height between 2 turns (in cm) in a
 

```
helix layer
if not present will default to
layerwidth
layerheight 0.161
```
- #number of helix turns per Layer
 

```
turns 3
```
- #number of helix layers
 

```
numlayers 1
```
- #the translate value...it will move every
 

```
point
by the value below
center 0,0,0
```

```

#the axis about which the coil will be
generated AROUND
either 'x', 'y', or 'z'
axis z

```

- Creating Your Own Shapes...

- These are the basic shapes available to you, however, should you wish to create your own shape, there is a way to write a function, then have BiotCoil coil recognize it from a ‘type’ argument in a Parameter set using the ‘Function Registration’ method.
- To register a function you simply need to write a function that is of the form

```

void MyNiftyCoil(Parameters &pset,
Vector< Vector< coord<> > &Coil);

```

- where the parameter set ‘pset’ contains the necessary input parameters as the above shapes do, and the `Vector< Vector< coord<> >` is the place where the points of the shape will be inputted via your calculation.
- In `main(int argc, char **argv)` (or some other function that gets called before you use this new coil) you will then need to register this new function by giving it a ‘type’ name.

```

BiotFunctions.insert("MyTypeName" ,
MyNiftyCoil); .

```

Once you have called this function, and Parameter set with the ‘type `MyTypeName`’ will look to this function to calculate the points of the coil.

- Your function should add ONE `Vector<coord<> >` to ‘Coil’ object. This is so that you can save this coil as a file with perhaps other coils in it to create a large master coil. Below is a pseudo code example of the desired properties

```

void MyNiftyCoil (Parameters &pset,
Vector< Vector< coord<> > &Coil)
{
//get the oodles of parameters like this
double mypar1=pset.getParamD("mypar");
int numpts=pset.getParamI("numpts");
//Now create a new Vector<coord<> > of
points
Vecotor<coord<> > tmV(numpts, 0);
//now fill it up...
for(int i=0;i< numpts;++i){
//filll me up...
}

//then add this new coil to the end of

```

```
the master list
Coil.push_back(tmV);
}
```

---

As stated before, this class simply acts a coil point interface;  
the Magnetic fields are calculated in the `Biot` and  
`MultiBiot` classes.

---

## BiotCoil::Public Vars

### Function:

**double amps**

### Input:

### Description:

This is the current running through the coil (in amps).

### Example Usage:

`BiotCoil myCoil;`  
`myCoil.amps=5.0; //5 amps in the coil`

## BiotCoil::Public Vars

**Function:** `Vector<Vector<coord>> > Coil;`

Input:

Description:  This container holds the coil points. It can have multiple sub coils (the Vector of Vector<coord>>).

Example

Usage: 

```
BiotCoil myCoil;
myCoil.amps=5.0; //5 amps in the coil
Vector<Vector<coord>> > pts;
for(int j=-1;j<3;++j){
pts.push_back(Vector<coord>>(0,0));
for(int i=0;i<1;i+=0.02){
pts[j].push_back(coord<>(i,j,i*i));
}
myCoil.Coil=pts; //set the coil points
```

## BiotCoil::Public Vars

**Function:**  **int nloops**

Input:

Description:  The number of fictitious wraps the coil makes. The effective current through an area would then be 'amps\*nloops.'

Example  `BiotCoil myCoil;`

Usage: `myCoil.amps=5.0; //5 amps in the coil`  
`myCoil.nloops=8; //8 loops in the coil`

## BiotCoil::constructor

### Function:

□ **BiotCoil()**

### Input:

□ void

### Description:

□ The empty constructor that sets the default values

npts=0  
type=""  
min=0  
max=0  
amps=0  
nloops=0  
Coli=empty

### Example Usage:

□ `BiotCoil myCoil;`

## BiotCoil::constructor

**Function:** `■ BiotCoil(std::string infile, std::string sec="coil");`

**Input:** `■ infile--> the name of the file to read`  
`sec--> the parameter section to obtain the coil info from (default="coil"). Setting sec="" will make the program read the global variables.`

**Description:** `■ Reads a Coil from the file 'infile' that contains a valid Parameter set with the section name 'sec.'`

**Example** `■ //Imagine an input file like so`

**Usage:** `-----`

```
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
```

## BiotCoil::constructor

**Function:** `■ BiotCoil(Parameters &pset, std::string sec="coil");`

**Input:** `■ pset--> a Parameter set  
sec--> the section in the parameter object to read from (default="coil")`

**Description:** `■ Creates a BiotCoil object from a previously declared parameter set, and reads the info from the section 'sec.'`

**Example** `■ //Imagine an input file like so`

**Usage:** `-----  
// mycoil{  
// type circle  
// numpts 4000  
// center -1,-1,-1  
// R 3  
// axis x  
// amps 2  
// loops 2  
//}  
-----  
  
//here is the file name  
std::string myfile="input.sim";  
Parameters pset(myfile);  
BiotCoil myCoil(pset, "mycoil");`

## BiotCoil::constructor

**Function:** `■ BiotCoil(const Vector<std::string> &pars, std::string sec="coil");`

**Input:** `■ pars--> a vector of strings  
sec--> the section in the parameter object to read from (default="coil")`

**Description:** `■ Creates a BiotCoil object from a set of strings, and reads the info from the section 'sec.'`

**Example** `■ //Imagine an input file like so`

**Usage:** `//-----  
// mycoil{  
// type circle  
// center -1,-1,-1  
// R 3  
// axis x  
// amps 2  
// loops 2  
//}  
//-----  
  
//here is the file name  
std::string myfile="input.sim";  
Parameters pset(myfile);  
BiotCoil myCoil(pset.section("mycoil"), "");`

## BiotCoil::constructor

**Function:**  `BiotCoil(std::string intype, int pts, double loops=1000, double inamps=2.5);`

**Input:**  `intype`--> the 'type' of shape (line, circle, etc)  
`pts`--> the number of points that define the coil  
`loops`--> the fictitious number of wraps for the coil  
`inamps`--> the current passed through the coil (in amps)

**Description:**  Creates a coil from the input (without the need for a parameter set).

**Example**  `//sadly this will compile, but generate`

**Usage:**  `// a bunch of errors when run`

`BiotCoil myCol("line", 4000);`

## BiotCoil::constructor

**Function:**  `BiotCoil(const Vector<coord<> > &pts, double loops=1000, double inamps=2.5);`

**Input:**  pts--> the Vector of (x,y,z) points that define a coil  
loops--> the number of wraps this coil makes  
inamps--> the current through the coil (in amps)

**Description:**  Creates a Boil object assuming the input Vector contains all the points in the shape.

**Example**  `Vector<coord<> > pts;`

**Usage:**  `for(int i=0;i<1;i+=0.02){  
 pts.push_back(coord<>(i,2*i,i*i));  
}  
BiotCoil myCoil(pts);`

## BiotCoil::constructor

**Function:**  `BiotCoil(const Vector<Vector<coord>> &pts, double loops=1000, double inamps=2.5);`

**Input:**  pts--> the Vector of Vectors of (x,y,z) points that define a coil  
loops--> the number of wraps this coil makes  
inamps--> the current through the coil (in amps)

**Description:**  Creates a Boil object assuming the input Vector contains all the points in the shape.

**Example**  `Vector<Vector<coord>> pts;`

**Usage:**  `for(int j=-1;j<3;++j){  
 pts.push_back(Vector<coord>>(0,0));  
 for(int i=0;i<1;i+=0.02){  
 pts[j].push_back(coord<>(i,j,i*i));  
 }  
 BiotCoil myCoil(pts);`

## BiotCoil::element extraction

### Function:

□ **coord<> constField();**

### Input:

□ void

### Description:

□ Returns the 'constant Field' value given ONLY if the type='constant.'

### Example Usage:

```
□ //Imagine an input file like so
//-----
// mycoil{
// type constant
// constField 4,70,8
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
cout<<myCoil.constField(); //prints [4,70,8]
```

## BiotCoil::element extraction

**Function:**  **std::string type();**

**Input:**  void

**Description:**  Returns the type that was read by the reader.

**Example Usage:**  //Imagine an input file like so

```
//-----
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
cout<<myCoil.type(); //will print "line"
```

## BiotCoil::assignments

**Function:** `coord<> constField(const coord<> &inF);`

**Input:** `inF-->` the new constant Field in GAUSS

**Description:**  `Sets the 'constant Field' value given ONLY if the type='constant.'`

**Example Usage:** `//Imagine an input file like so`

```
//-----
// mycoil{
// type constant
// constField 4,70,8
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
myCoil.constField(coord<>(3,8,90)); //sets the field
```

## BiotCoil::IO

### Function:

□ **void print(std::ostream &out=cout);**

### Input:

□ out--> an output stream

### Description:

□ Prints a valid Parameter set representation of the input coil.

### Example Usage:

```
□ //Imagine an input file like so
//-----
// mycoil{
// type constant
// constField 4,70,8
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
myCoil.print();
//prints
//#BiotCoil..
//mycoil{
// type constant
// constField 4,70,8
//}
```

## BiotCoil::IO

**Function:** `void read(std::string in, std::string sec="coil");`

**Input:**

- ❑ infile--> the name of the file to read
- ❑ sec--> the parameter section to obtain the coil info from (default="coil"). Setting sec="" will make the program read the global variables.

**Description:**

- ❑ Reads the boitcoil from a file give the name 'in' from the valid parameter section 'sec'. This is like the constructor of the same form, except this re-reads the Coil.

**Example**    `//Imagine an input file like so`

**Usage:**

```
//-----
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil;
myCoil.read(myfile, "mycoil");
```

## BiotCoil::IO

**Function:**  **void read(Parameters &in, std::string sec="coil");**

**Input:**  pset--> a Parameter set  
sec--> the section in the parameter object to read from (default="coil")

**Description:**  Reads a coil from a previously declared parameter set, and reads the info from the section 'sec'.

**Example Usage:**  //Imagine an input file like so

```
//-----
// mycoil{
// type circle
// center -1,-1,-1
// R 3
// axis x
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
Parameters pset(myfile);
BiotCoil myCoil;
myCoil.read(pset, "mycoil");
```

## BiotCoil::IO

**Function:** `void read(const Vector<std::string> &in, std::string sec="coil");`

**Input:**

- pars--> a vector of strings
- sec--> the section in the parameter object to read from (default="coil")

**Description:** □ Reads a coil object from a set of strings, and reads the info from the section 'sec'.

**Example Usage:** □ //Imagine an input file like so

```
//-----
// mycoil{
// type circle
// numpts 4000
// center -1,-1,-1
// R 3
// axis x
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
Parameters pset(myfile);
pset.addSection("mycoil");
BiotCoil myCoil;
myCoil.read(pset.section("mycoil"), "");
```

## BiotCoil::IO

**Function:**  **void read();**

**Input:**  void

**Description:**  Simply Re-Reads the Parameter set given in the constructor or an previous read function.

**Example Usage:**  //Imagine an input file like so

```
//-----
// mycoil{
// type circle
// numpts 4000
// center -1,-1,-1
// R 3
// axis x
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
Parameters pset(myfile);
BiotCoil myCoil;
myCoil.read(pset, "mycoil");
myCoil.read(); //read it again
```

## BiotCoil::IO

**Function:**  **void readShape(std::string in);**  
 **void readShape(std::fstream &in);**

**Input:**  in--> The file name (std::string) OR an input stream (std::fstream)

**Description:**  Reads a file in the 'BiotCoil' format. It does not write the Parameter set, but it reads the format as generated by 'writeShape'

```
START BiotCoil
amps {num}
loops {num}
{x,y,z}
{x,y,z}
....
```

```
END BiotCoil
```

**Example**  //Imagine an input file like so

**Usage:**

```
//-----
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
myCoil.writeShape("outfile");
//will print
//START BiotCoil
//amps 2
//loops 2
// -1 -1 -1
//....
// 1 1 1
//END BiotCoil
myCoil.readShape("outfile"); //ill read that file
```

## BiotCoil::IO

**Function:**  **void writeShape(std::string &out);**  
 **void writeShape(std::fstream &out);**

**Input:**  out--> The file name (std::string) OR an ouput stream (std::fstream)

**Description:**  Writes a file in the 'BiotCoil' format. It does not write the Parameter set, but it writes the calculated points with the current values of the amps and nloops. It has the form

```
START BiotCoil
amps {num}
loops {num}
{x,y,z}
{x,y,z}
....
```

```
END BiotCoil
```

**Example**  //Imagine an input file like so

**Usage:**  //-----

```
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
// }
//-----
```

```
//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
myCoil.writeShape("outfile");
//will print
//START BiotCoil
//amps 2
//loops 2
// -1 -1 -1
//....
// 1 1 1
//END BiotCoil
```

## BiotCoil::other

**Function:**  coord<> min();  
 coord<> max();

**Input:**  void

**Description:**  These return the min (or max) extent of the coil in all three directions. Most likely this will NOT be a point in the coil .

**Example**  //Imagine an input file like so

**Usage:**  //-----

```
// mycoil{
// type line
// numpts 4000
// begin -1,1,1
// end 1,-1,2
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
cout<<myCoil.min(); //outputs [-1,-1,1]
```

## BiotCoil::other

**Function:** `int size();`

**Input:** `void`

**Description:** `>Returns the size of the coil. It return the TOTAL number of points (i.e. a Sum of all the sub coils in the 'Coil' variable).`

**Example** `//Imagine an input file like so`

**Usage:** `-----`

```
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
cout<<myCoil.size(); //should print 4000
```

## BiotCoil::other

### Function:

```
□ void translate(const coord<> &dir);
 void translate(double x, double y, double z);
```

### Input:

```
□ dir--> a coord of x,y,z lengths to move the coil
 x,y,z--> the lengths to move the coil
```

### Description:

```
□ Moves the entire coil by the lengths specified in the input (x,y,z).
```

### Example Usage:

```
□ //Imagine an input file like so
//-----
// mycoil{
// type line
// numpts 4000
// begin -1,-1,-1
// end 1,1,1
// amps 2
// loops 2
//}
//-----

//here is the file name
std::string myfile="input.sim";
BiotCoil myCoil(myfile, "mycoil");
//move the entire coil by 5 in x, and 6 along y and z
myCoil.translate(5,6,6);
```

**--Public Vars**[--- Bfield](#)[--- Controller](#)**--Constructors**[--- Biot](#)[--- Biot](#)[--- Biot](#)[--- Biot](#)[--- Biot](#)[--- Biot](#)**--Assignments**[--- operator=](#)[--- setGrid](#)**--IO**[--- read](#)[--- read](#)[--- write](#)[--- writeMatlab](#)**--Other Functions**[--- calculateField](#)**Examples**[--- True Helmholtz](#)

template&lt;class Grid\_t&gt;

class Biot:

public BiotCoil

{....

---

This class contains the algorithms necessary to calculate Magnetic Fields on specific grid points using a coil as defined in BiotCoil object.

It is useful for single coils only. The class 'MultiBiot' calculates fields generated by multiple coils, but does not store the field for each coil. This class stores the magnetic fields for the individual coil. Thus if you need just the field, and are not particular from which coil it came from, use the MultiBiot class. However, if you need to know the coil generated by each individual coil, then use a Vector of this class.

---

## Biot::Public Vars

**Function:** `Vector<coord> > Bfield;`

Input:

Description:  This holds the calculated magnetic field. Each point matches the order of the Grid Vector in the class.

Example    `Parameters pset("myfile");  
pset.addSection("mycoil");`

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//print out the grid points and the field
XYZshape<XYZfull>::iterator it(g2);
while(it){
cout<<it.Point()<<" "<<mycoil.Bfield[it.curpos()]<<endl;
}
```

## Biot::Public Vars

**Function:** □ **MPIcontroller Controller**

Input: □

Description: □ This is the Biot's MPI parallel controller. If this is not set, the magnetic field calculation will NOT be performed in parallel.

Example □ Parameters pset( "myfile" );

Usage: pset.addSection( "mycoil" );

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroler to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();
```

## Biot::constructor

### Function:

□ **Biot()**

### Input:

□ void

### Description:

□ Empty constructor. Sets up an empty coil and an empty grid.

### Example Usage:

□ `Biot<XYZshape<XYZfull> > coil;`

## Biot::constructor

**Function:** `■ Biot(std::string fname, std::string sec="coil");`

Input: `■ fname--> a file name`

`sec--> the coil section you wish to load`

Description: `■ Attempts to read the coil named 'sec' from the parameters from a file names 'fname.'` It also attempts to read the grid.

Example `■ //The input file`

Usage: `//`

```
// coil{
// type constant
// constField 4,50,8
//}
//
// grid{
// min -1, -1, -1
// max 1, 1, 1
// dim 10, 10,10
//}
```

```
Biot<XYZshape<XYZfull> > myField("infile");
```

## Biot::constructor

**Function:** `■ Biot(Parameters &pset, std::string sec="coil");`

**Input:** `■ pset--> a Parameters object`  
`sec--> the coil section you wish to load`

**Description:** `■ Attempts to read the coil named 'sec' from the parameters from a file names 'fname.' It also attempts to read the grid.`

**Example** `■ //The input file`

**Usage:**

```
//
// TheCoil{
// type constant
// constField 4,50,8
//}
//
// grid{
// min -1, -1, -1
// max 1, 1, 1
// dim 10, 10,10
//}
Parameters pset("infile");
//this will read the 'grid' section as well
Biot<XYZshape<XYZfull> > myField(pset, "TheCoil");
```

## Biot::constructor

**Function:** `■ Biot(Grid_t &ingrid, Parameters &pset, std::string sec="coil"):`

Input: `■ ingrid--> the grid`

`pset--> the Parameter file`

`sec--> the desired coil section`

Description: `■ Attempts to read the coil named 'sec' from the parameters from a file names 'fname.' It does NOT read the grid from the parameter file.`

Example `■ Parameters pset("myfile");`

Usage: `pset.addSection("mycoil");`

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZShape<XYZfull> > mycoil(g2,pset, "mycoil");
```

## Biot::constructor

**Function:** `■ Biot(const BiotCoil &bc, const Grid_t &gg):`

Input: `■ bc--> a BiotCoil object`  
`gg--> a Grid object`

Description: `■ Creates a Biot object from a already declared BiotCoil and a grid.`

Example `■ Parameters pset("myfile");`

Usage: `pset.addSection("mycoil");`

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

BiotCoil coil(pset, "mycoil");
//set up our Biot field calculator class
Biot<XYZshape<> > mycoil(coil,g2);
```

## Biot::constructor

|                       |                                                                                                                                                                                                    |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b>      | <ul style="list-style-type: none"><li>□ <b>Biot(const Biot &amp;cp);</b></li></ul>                                                                                                                 |
| <b>Input:</b>         | <ul style="list-style-type: none"><li>□ cp--&gt; another Biot with the same grid type</li></ul>                                                                                                    |
| <b>Description:</b>   | <ul style="list-style-type: none"><li>□ The copy constructor.</li></ul>                                                                                                                            |
| <b>Example Usage:</b> | <ul style="list-style-type: none"><li>□ <code>Biot&lt;XYZshape&lt;XYZfull&gt; &gt; myField("infile");</code></li><li>□ <code>Biot&lt;XYZshape&lt;XYZfull&gt; &gt; ccopy(myField);</code></li></ul> |

## Biot::assignments

**Function:** `void operator=(const Biot<Grid_t> &rhs)`

**Input:** `rhs-->` a Biot object

**Description:** `Assigns one Biot of the same grid type to another.`

**Example** `Parameters pset("myfile");`

**Usage:** `pset.addSection("mycoil");`

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");
Biot<XYZshape<XYZfull> > mycoil2=mycoil;
```

## Biot::assignments

**Function:** `void setGrid(const Grid_t &ingr);`

**Input:** `ingr-->` a object of the type 'Grid\_t'

**Description:** `Sets the grid. It must be of the same type as Grid_t.`

**Example** `Parameters pset("myfile");`

**Usage:** `pset.addSection("mycoil");`

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(pset, "mycoil");
mycoil.setGrid(g2);
```

## Biot::IO

**Function:**  `void read(std::string &fname, bool readgrid=true);`  
`void read(Parameters &pset, bool readgrid=true);`

**Input:**  fname--> a file name  
pset--> a previously declared parameter set  
readgrid--> if true will try to read a 'grid' section from the Parameters

**Description:**  Reads a file (if a string) or a parameter set. However, this assumes that you have set the section in the BiotCoil superclass (using the Biot(pset, section) type constructors or by calling 'read(pset, section)'). If readgrid is true then it will try to read the section 'grid' from the Parameter input.

**Example**  `//The input file`

**Usage:** `//`  
`// coil{`  
`// type constant`  
`// constField 4,50,8`  
`//}`  
`//`  
`// grid{`  
`// min -1, -1, -1`  
`// max 1, 1, 1`  
`// dim 10, 10,10`  
`//}`  
`Biot<XYZshape<XYZfull> > coil;`  
`Parameters pset("infile");`  
`coil.read(pset, "coil"); //read the coil`  
`coil.read(pset); //read the grid`

## Biot::IO

**Function:**  **void read(std::fstream &out);**

**Input:**  out--> an i/o stream (opened with ios::in flags)

**Description:**  Reads a file in the 'BiotCoil' format. It does not write the Parameter set, but it reads the format as generated by 'BiotCoil::writeShape'

If there is a magnetic field present it then attempts to read the magnetic field. Below is the format.

-----  
START BiotCoil

amps {num}

loops {num}

{x,y,z}

{x,y,z}

....

END BiotCoil

-----  
START BiotCoilBfield

size 1000

{bx,by,bz}

{bx,by,bz}

....

END BiotCoilBfield

**Example**

Parameters pset("myfile");  
Usage: pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.write(out);

//read in a new object from the old one
Biot<> mycoil2;
ifstream in("outfile.biot");
```

```
mycoil.read(in);
```

## Biot::IO

**Function:**  **void write(std::string fname);**  
**void write(std::fstream &out);**

**Input:**  fname--> a file name to dump the info  
out--> a previously declared output stream

**Description:**  Writes a Biot object as both the coil and the magnetic field (if there is any). In the format...

```

START BiotCoil
amps {num}
loops {num}
{x,y,z}
{x,y,z}
....
END BiotCoil

START BiotCoilBfield
size 1000
{bx,by,bz}
{bx,by,bz}
....
END BiotCoilBfield

```

**Example**  Parameters pset( "myfile" );  
**Usage:** pset.addSection( "mycoil" );

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.write(out);
```

## Biot::IO

**Function:**

- ❑ `void writeMatlab(std::string fname);`
- ❑ `void writeMatlab(matstream &out);`

**Input:**

- ❑ `fname`--> an output file name
- ❑ `out`--> a previously declared matlab output stream

**Description:**

- ❑ Writes the data inside the object to several matlab variables to allow the matlab script 'plotmag' (in the folder 'matlabfunc' in the distribution) to plot the magnetic fields and coils. The data format inside this class is not easily transcribed into a plotable matlab format, thus the data is reordered and reorganized as follows.

B0 - the Bx field (a 1xN double array)

B1 - the By field (a 1xN double array)

B2 - the Bz field (a 1xN double array)

P0 - the coil's x point (a 1xN array)

P1 - the coil's y point (a 1xN array)

P2 - the coil's z point (a 1xN array)

pdiv - the indexs inside (P0,P1,P2) where new coils start

grid0 - the grid's x points (a 1xN array)

grid1 - the grid's y points (a 1xN array)

grid2 - the grid's z points (a 1xN array)

N - the number of points total (constant)

r - width of the grid (grid.max-grid-min) (a 1x3 array)

res - the resolution (the dx, dy, dz) (a 1x3 array)

-----  
The matlab function the reassembles the data into valid  
3D arrays for each direction.

**Example**

```
❑ //set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));
```

```
//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());
```

```
//set up our Biot field calculator class
Biot<XYZshape<> > mycoil(g2,pset, "mycoil");
```

```
//set the MPIcontroler to be MPIworld
mycoil.Controller=MPIworld;
```

```
//calculate the field
mycoil.calculateField();
```

```
//dump data into a matlab file
```

```
mycoil.writeMatlab("out.mat");
```

## Biot::other

**Function:** `bool calculateField();`

Input: `void`

Description: `Calculates the magnetic field on each grid point from the coil. It will calculate it in parallel if Controller is set. Returns false if something failed.`

Example `//set up our base rectangular grids`

Usage: `Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1), coord<int>(10,10,10));`

```
//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());
```

```
//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");
```

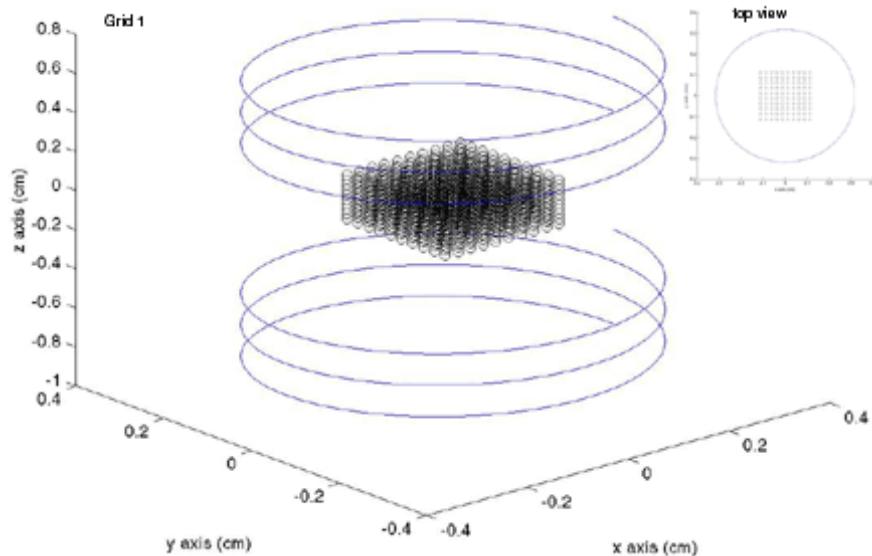
```
//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;
```

```
//calculate the field
mycoil.calculateField();
```

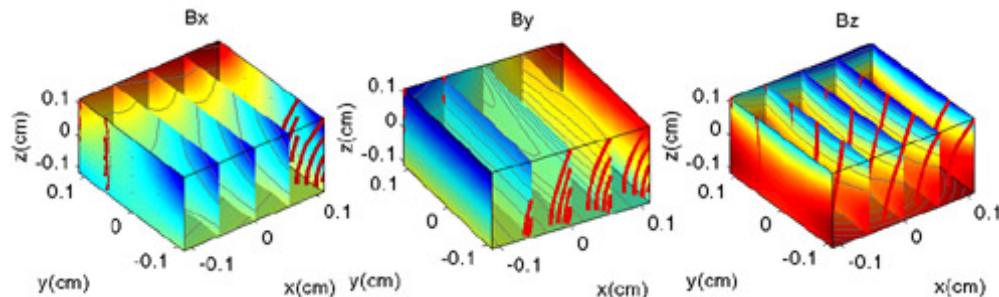
Here is the field from a split solonoid. A split solonoid is simply a close view of a true helmholtz pair. The code calculates 2 different views of the field, one is a larger perspective to show the fringe fields, and another to show the field inside the cavity.

- There is a Matlab Function called **plotmag** The plots these figures from the generated matlab files. This file is in the folder 'matlabfunc' in the distribution top directory.
- Here is the generated data from the First Grid

- The Grid and coil

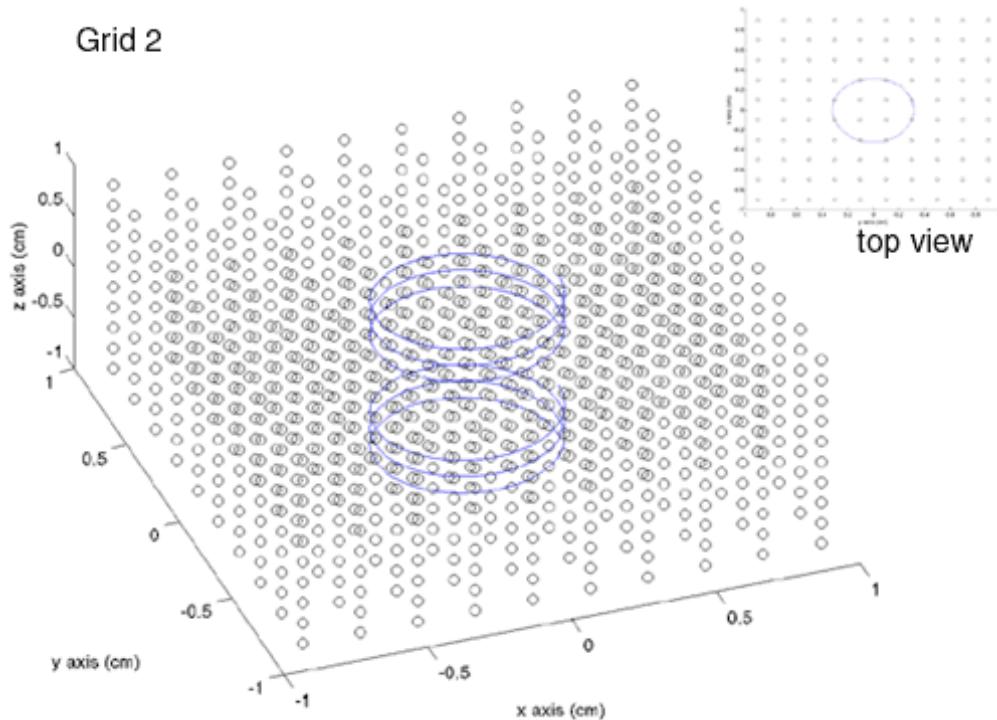


- The Magnetic Fields along each axis

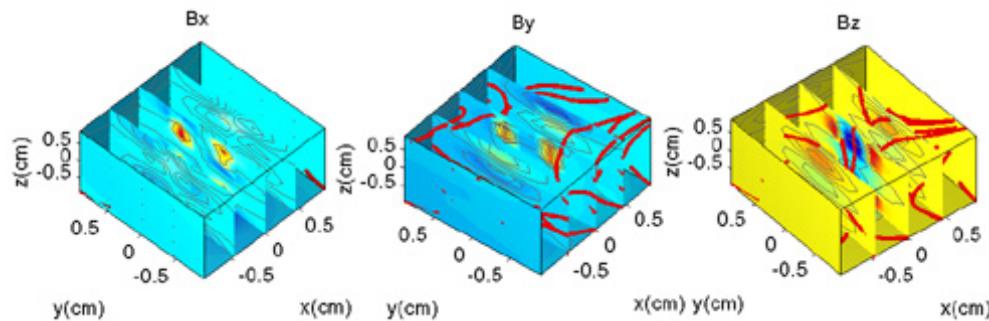


- Here is the generated data from the Second Grid
- The Grid and coil

## Grid 2



- The Magnetic Fields along each axis




---

```
//It takes in this as the input file
-----truh.sim-----

#the coil section we wish to use...
section helmz
 #our 'split-solenoid'
 helmz{
 type truehelmholtz
 loops 1
 amps 3
 numpts 8000
 # radius (cm)
 R 0.3175
 #distance b/w the two coils (cm)
 length 1.09
 #width between 2 helix layers
 layerwidth 0.161
 #height between 2 turns in a helix layer
 # if not present will default to layerwidth
 layerheight 0.161
 #number of helix turns
 turns 3
 #number of helix layers
 numlayers 1
 axis z
 }
 #number of grids to calculate the field over
```

```

numGrids 2
#our rectangular grid dimensions
grid1{
 min -0.125, -0.125, -0.125
 max 0.125, 0.125, 0.125
 dim 10, 10, 10
}
#our rectangular grid dimensions
grid2{
 min -1, -1, -1
 max 1, 1, 1
 dim 10, 10, 10
}
-----truh.sim-----
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

//The main runner for calculating magnetic fields
//over grids a set number of grid points

int main(int argc, char **argv)
{

//start MPI
 MPIworld.start(argc, argv);
 std::string parse="";
 int q=1;
//get the parameter file
 if(MPIworld.master())
 query_parameter(argc, argv, q++, "Enter Parmeter set file name:", parse);

//distribute the file name to all the nodes
 MPIworld.scatter(parse);

//decalare our Parameter set
 Parameters pset(parse);

//get the desired coil to calculate the field over
 std::string choose=pset.getParamS("section");

//add this section to the parameter set
 pset.addSection(choose);

//a typedef to make our grid typing easier
 typedef XYZshape<XYZfull> TheGrid;

//get the number of grids we wish to calculate the field over
 int numGrids=pset.getParamI("numGrids");

//the grids sectiosn will look like 'grid1', 'grid2'
 std::string baseGname="grid";

 for(int i=1;i<=numGrids;++i)
 {
 std::string Gname=baseGname+itost(i);
//add the grid section
 pset.addSection(Gname);

//set up our base rectangular grids
 Grid<UniformGrid> g1(pset.getParamCoordD("min", Gname),
 pset.getParamCoordD("max", Gname),
 pset.getParamCoordI("dim", Gname));

//set up thte master shape
 TheGrid g2(g1, XYZfull());

```

```
//set up our Biot field calculator class
Biot<TheGrid> mycoil(g2,pset, choose);

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//dump out the info to files
if(MPIworld.master()){
 mycoil.writeMatlab(pset.getParamS("matout", "params", false, "field"+itost(i)+".mat"));
 mycoil.write(pset.getParamS("textout", "params", false, "shape"+itost(i)+".boit"));
}
}

//end out MPI session
MPIworld.end();

}
```

---

**--Public Vars**[--- Bfield](#)[--- Controller](#)[--- grid](#)**--Constructors**[--- MultiBiot](#)[--- MultiBiot](#)[--- MultiBiot](#)**--IO**[--- read](#)[--- read](#)[--- readShape](#)[--- write](#)[--- writeMatlab](#)[--- writeShape](#)**--Other Functions**[--- averageField](#)[--- calculateField](#)[--- maxField](#)**Examples**[--- generic MultiBiot](#)

```
template<class Grid_t=XYZshape<XYZfull> >
class MultiBiot{...
```

---

The input file for a MultiBiot look like the one below. You can have as many 'subcoil' sections (up to 100000). If you have a 'grid' section preset, the class will also read that section in order to create the grid. The subcoil sections start at 1. It will stop looking for sections when it cannot find the next subcoil in the master section it will stop looking for subsections (even if there are more present). If you have a subcoil1 and a subcoil3 section with no subcoil2 it will stop reading at subcoil1.

```
MyCoil{
 subcoil1{
 type line
 begin 0,3,5
 end 1,1,1
 amps 3
 loops 1
 numpts 5000
 }

 subcoil2{
 type line
 begin 1,1,1
 end 2,2,2
 amps 3
 loops 1
 numpts 5000
 }

 grid{
 min -1,-1,-1
 max 1,1,1
 dim 10,10,10
 }
}
```

---

## MultiBiot::Public Vars

**Function:**  **Vector<coord<> > Bfield;**

Input:

Description:  This holds the calculated magnetic field. Each point matches the order of the Grid Vector in the class.

Example  Parameters pset("myfile");

Usage: pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
MultiBiot<> mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//print out the grid points and the field
XYZshape<XYZfull>::iterator it(g2);
while(it){
cout<<it.Point()<<" "<<mycoil.Bfield[it.curpos()]<<endl;
}
```

## MultiBiot::Public Vars

### Function: MPIcontroller Controller

Input:

Description:  This is the MultiBiot's MPI parallel controller. If this is not set, the magnetic field calculation will NOT be performed in parallel.

Example  Parameters pset("myfile");  
Usage: pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
MultiBiot<> mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();
```

## MultiBiot::Public Vars

**Function:** `Grid_t grid;`

Input: `grid`

Description: `grid` The grid to calculate the magnetic field.

Example `MultiBiot<> coil;`

Usage: `//set up our base rectangular grids`

```
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));
```

```
//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());
```

```
//set the grid
coil.grid=g2;
```

## MultiBiot::constructor

**Function:**  **MultiBiot(Parameters &pset, std::string sec="coil");**

Input:  pset--> a Parameter set

sec--> the master coil section name

Description:  Attempts to read the coil named 'sec' from the parameters from a file names 'fname.' It also attempts to read the grid. If no grid is present in the Parameter set then you will need to specify one yourself.

Example  //The input file

Usage: //

```
// TheCoil{
// subcoil1{
// type constant
// constField 4,50,8
// }
// subcoil2{
// type line
// begin 1,1,1
// end 2,2,2
// amps 3
// loops 1
// numpts 5000
// }
//}
//
// grid{
// min -1, -1, -1
// max 1, 1, 1
// dim 10, 10,10
//}
Parameters pset("infile");
//this will read the 'grid' section as well
MultiBiot<> myField(pset, "TheCoil");
```

## MultiBiot::constructor

**Function:**  `MultiBiot(Grid_t &ingrid, Parameters &pset, std::string sec="coil");`

**Input:**  `ingrid-->` the grid  
`pset-->` the Parameter file  
`sec-->` the desired master coil section

**Description:**  Attempts to read the multi-coil named 'sec' from the parameters set 'pset.' It does NOT read the grid from the parameter file.

**Example**  `//The input file`

**Usage:**

```
//
// TheCoil{
// subcoil1{
// type constant
// constField 4,50,8
// }
// subcoil2{
// type line
// begin 1,1,1
// end 2,2,2
// amps 3
// loops 1
// numpts 5000
// }
//}
//
// grid{
// min -1, -1, -1
// max 1, 1, 1
// dim 10, 10,10
//}

Parameters pset("myfile");

//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
MultiBiot<> mycoil(g2,pset, "TheCoil");
```

## MultiBiot::constructor

**Function:**  **MultiBiot()**

**Input:**  void

**Description:**  Empty constructor. Sets all variables to empty or 0.

**Example Usage:**  MultiBiot<> myCoil;

## MultiBiot::IO

**Function:** `void read(Parameters &pset, std::string sec="", bool readgrid=true);`

Input: `pset`--> a previously declared parameter set

`sec`--> the MASTER section of the coil

`readgrid`--> if true will try to read a 'grid' section from the Parameters

Description: `Reads a multi coil object from a parameter set. It will attempt to read the section 'grid' as well if 'readgrid' is true.`

Example `//The input file`

Usage: `//`

```
// MyCoil{
// subcoil1{
// type line
// begin 0,3,5
// end 1,1,1
// amps 3
// loops 1
// numpts 5000
// }
//
// subcoil2{
// type line
// begin 1,1,1
// end 2,2,2
// amps 3
// loops 1
// numpts 5000
// }
//}
//
// grid{
// min -1, -1, -1
// max 1, 1, 1
// dim 10, 10,10
// }
```

```
MultiBiot<> coil;
Parameters pset("infile");
coil.read(pset, "MyCoil"); //read the coil and grid
```

## MultiBiot::IO

**Function:**  **void read(std::string fname);**  
**void read(std::fstream &file);**

**Input:**  fname--> the name of the text file  
file--> a previously declared input stream

**Description:**  Reads BOTH the shape and the Magnetic Field in the following format...

-----  
START MultiBiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

....

END MultiBiotCoil

START MultiBiotCoilBfield

{bx by bz}

....

END MultiBiotCoilBfield

**Example**

Parameters pset("myfile");  
pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;
```

```
//calculate the field
mycoil.calculateField();

//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.write(out);

MultiBiot<> mycoil2;
ifstream in("outfile");
mycoil2.read(in);
```

## MultiBiot::IO

**Function:**  **void readShape(std::string fname);**  
**void readShape(std::fstream &file);**

**Input:**  fname--> the name of the text file  
file--> a previously declared input stream

**Description:**  Reads the shape of the multi-coil from the follow format.

-----  
START MultiBiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

END MultiBiotCoil

-----

**Example**  Parameters pset("myfile");

**Usage:**  pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
MultiBiot<> mycoil(g2,pset, "mycoil");
//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.writeShape(out);

//read in a new object from the old one
MultiBiot<> mycoil2;
ifstream in("outfile.biot");
mycoil.readShape(in);
```



## MultiBiot::IO

**Function:**  `void write(std::string fname);`  
`void write(std::fstream &file);`

**Input:**  `fname`--> the name of the text file  
`file`--> a previously declared output stream

**Description:**  Writes BOTH the shape and the Magnetic Field (if calculated or previously read in) in the following format...

-----  
START MultiBiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

....

END MultiBiotCoil

START MultiBiotCoilBfield

{bx by bz}

....

END MultiBiotCoilBfield

-----

Example     Parameters pset("myfile");  
Usage:      pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();

//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.write(out);
```

## MultiBiot::IO

**Function:**  `void writeMatlab(std::string fname);`  
`void writeMatlab(matstream &matout);`

**Input:**  `fname`--> an output file name  
`out`--> a previously declared matlab output stream

**Description:**  Writes the data inside the object to several matlab variables to allow the matlab script 'plotmag' (in the folder 'matlabfunc' in the distribution) to plot the magnetic fields and coils. The data format inside this class is not easily transcribed into a plotable matlab format, thus the data is reordered and reorganized as follows.

B0 - the Bx field (a 1xN double array)

B1 - the By field (a 1xN double array)

B2 - the Bz field (a 1xN double array)

P0 - the coil's x point (a 1xN array)

P1 - the coil's y point (a 1xN array)

P2 - the coil's z point (a 1xN array)

pdiv - the indexs inside (P0,P1,P2) where new coils start

grid0 - the grid's x points (a 1xN array)

grid1 - the grid's y points (a 1xN array)

grid2 - the grid's z points (a 1xN array)

N - the number of points total (constant)

r - width of the grid (grid.max-grid-min) (a 1x3 array)

res - the resolution (the dx, dy, dz) (a 1x3 array)

**Example**  `//set up our base rectangular grids`

**Usage:**  `Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1), coord<int>(10,10,10));`

`//set up thte master shape`

`XYZshape<XYZfull> g2(g1, XYZfull());`

`//set up our Biot field calculator class`

`MultiBiot<> mycoil(g2,pset, "mycoil");`

`//set the MPIcontroler to be MPIworld`

`mycoil.Controller=MPIworld;`

`//calculate the field`

`mycoil.calculateField();`

`//dump data into a matlab file`

`mycoil.writeMatlab("out.mat");`

## MultiBiot::IO

**Function:**  **void writeShape(std::string out);**  
**void writeShape(std::fstream &out);**

**Input:**  fname--> the name of the text file  
file--> a previously declared output stream

**Description:**  Writes the shape of the multi-coil to the follow format.

-----  
START MultiBiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

START BiotCoil

amps {num}

loops {num}

{x y z}

{x y z}

....

END BiotCoil

END MultiBiotCoil

-----

**Example**  Parameters pset("myfile");  
**Usage:**  pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up thte master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Biot field calculator class
Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");

//write the field out
// in the 'biot' format
ostream out("outfile.biot");
mycoil.writeShape(out);
```

## MultiBiot::other

**Function:** `coord<> averageField(coord<int> which=(1,1,1))`

**Input:** `which-->` a coord<int> the specifies the direction of the max. The default is all directions.

**Description:** `which` Returns the average field value along each direction where which[i]!=0. To obtain the maximum field along z only set which=(0,0,1).

**Example** `//set up our Biot field calculator class`

**Usage:** `Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");  
//calculate the field  
mycoil.calculateField();`

`//the max field along all 3 directions`

`cout<<mycoil.averageField()<<endl;`

`//the max field along the z-axis`

`cout<<mycoil.averageField(coord<int>(0,0,1));`

## MultiBiot::other

**Function:**  **void calculateField();**

Input:  void

Description:  Calculates the magnetic field at each grid point from the multi-coil set. If 'Controller' is set then it will calculate it in parallel.

Example  Parameters pset("myfile");

Usage: pset.addSection("mycoil");

```
//set up our base rectangular grids
Grid<UniformGrid> g1(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(10,10,10));

//set up the master shape
XYZshape<XYZfull> g2(g1, XYZfull());

//set up our Multi-Biot field calculator class
MultiBiot< > mycoil(g2,pset, "mycoil");

//set the MPIcontroller to be MPIworld
mycoil.Controller=MPIworld;

//calculate the field
mycoil.calculateField();
```

## MultiBiot::other

**Function:** `coord<> maxField(coord<int> which=(1,1,1))`

**Input:** `which-->` a coord<int> the specifies the direction of the max. The default is all directions.

**Description:** `>Returns the maximum field along each direction where which[i]!=0. To obtain the maximum field along z only set which=(0,0,1).`

**Example** `//set up our Biot field calculator class`

**Usage:** `Biot<XYZshape<XYZfull> > mycoil(g2,pset, "mycoil");`

`//calculate the field`

`mycoil.calculateField();`

`//the max field along all 3 directions`

`cout<<mycoil.maxField()<<endl;`

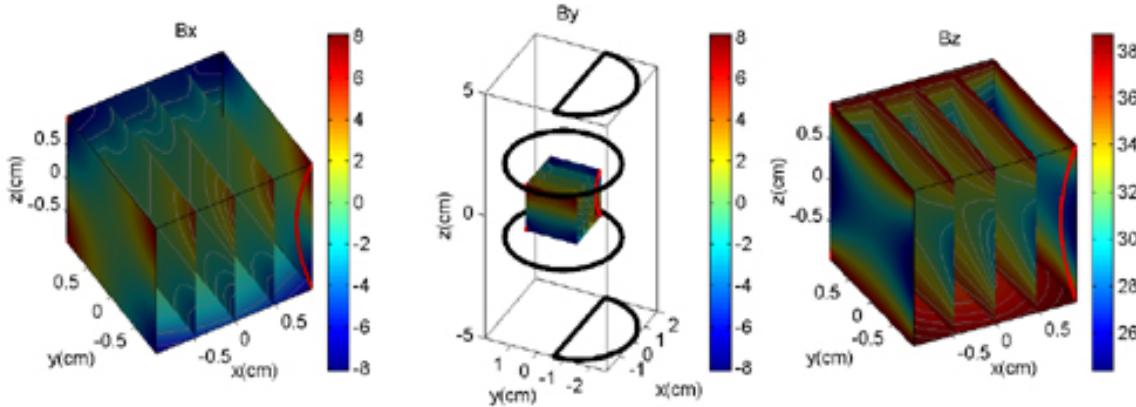
`//the max field along the z-axis`

`cout<<mycoil.maxField(coord<int>(0,0,1));`

Here is an example that goes through BOTH how to add a new coil function to the registration list, and how to use MultiBiot to calculate fields.

The type added here is a D-circle (a circle with a flatten side like a 'D'). The code for this example is long because the 'Dcircle' function is quite involved.

Below is the data as generated by the program plotted by **plotmag** found in the **matlabfunc** directory of the distribution.



```
//***Here is an example input file
-----input.sim-----
MyCoil{
 subcoil1{
 type helmholtz
 loops 25
 amps -4
 numpts 4000
 R 4
 length 3
 axis z
 }

 subcoil2{
 type Dcircle
 loops 1
 amps 2

 numpts 2000
 R 2

 #start theta of line section
 theta1 0
 theta2 180

 axis x
 center 0,-.6,5
 }

 subcoil3{
 type Dcircle
 loops 1
 amps 2
 }
}
```

```

numpts 2000
R 2

#start theta of line section
theta1 0
theta2 180
axis x
center 0,-.6,-5
}

}

Cube
grid{
 min -1,-1,-1
 max 1,1,1
 dim 20,20,20
}

params{
 #which magnetic field section to use
 section MyCoil

 #output text file name
 textout shape.biot
 #output matlab file name
 matout field.mat

}

-----input.sim-----
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

//***** THE D_CIRCLE BIOT FUNCTION*****
//the D circle (or 'D') is basically a cirlce with a flattened surface
// from theta=theta1...theta2
void Biot_Dcircle(Parameters &pset, Vector<Vector<coord>> &Coil)
{
 int i;
 double R=pset.getParamD("R");
 double theta1=pset.getParamD("theta1")*PI/180.0;
 double theta2=pset.getParamD("theta2")*PI/180.0;
 double startTH=pset.getParamD("startTheta", "", false, 0)*PI/180.0;
 double endTH=pset.getParamD("endTheta", "", false, 0)*PI/180.0;

 theta1=fmod(theta1, PI2);
 startTH=fmod(startTH, PI2);
 theta2=fmod(theta2, PI2);
 if(theta1>theta2) swap_(theta1, theta2);

 int numpts=pset.getParamI("numpts");
 char axis=pset.getParamC("axis", "", false, 'z');
 coord<> center=pset.getParamCoordD("center", "", ',', false);

 double angle=2.0*PI/double((numpts-1));
}

```

```

Coil.resize(1, Vector<coord> >(numpts,0.0));

double curang=startTH, endAng=0.0;
bool got1=false, got2=false, runline=false, ToAdd=true;
int linect=0, onPt=0;
coord<> begin, end, div, ston;

for (i=0;i<numpts;i++){
//number of points we've calculated
// if(curang>=startTH && curang<fmod(angle+startTH,PI2))
//our angle is within the part we want a line from
 if(curang>=theta1 && curang<theta2)
 {
//we just got here so snage the begining point
 if(!got1)
 {
 got1=true; //did the first point
 got2=false; //not found the last point
//do NOT caluclated the line until we get the end point
 runline=false;
 linect=0; //restart the line counter
 switch(axis)
 {
 case 'x': begin(0.0,
 R*sin(curang),
 R*cos(curang));
 break;
 case 'y': begin(R*cos(curang),
 0.0,
 R*sin(curang));
 break;
 default: begin(R*cos(curang),
 R*sin(curang),
 0.0);
 break;
 }
 linect++;
 ToAdd=false; //do not add an cylider bits
 }
 }
//found the upper bound
 if(got1 && !got2 && curang>=theta2)
 {
 got2=true; //found the end
 runline=true; //now calc the line
 //grab the end point
 endAng=curang;
 switch(axis)
 {
 case 'x': end(0.0,
 R*sin(curang),
 R*cos(curang));
 break;
 case 'y': end(R*cos(curang),
 0.0,
 R*sin(curang));
 break;
 }
 }
}

```

```

 default: end(R*cos(curang),
 R*sin(curang),
 0.0);
 break;
 }
 ston=begin+center;
 onPt=i;
 div=(end-begin)/double(linect);
}

//add the helix part if we can
if(ToAdd){
 switch(axis){
 case 'x':
 Coil[0][i][0]=0.0;
 Coil[0][i][1]=R*sin(curang);
 Coil[0][i][2]=R*cos(curang);
 break;
 case 'y':
 Coil[0][i][0]=R*cos(curang);
 Coil[0][i][1]=0.0;
 Coil[0][i][2]=R*sin(curang);
 break;
 case 'z':
 default:
 Coil[0][i][0]=R*cos(curang);
 Coil[0][i][1]=R*sin(curang);
 Coil[0][i][2]=0.0;
 break;
 }
 Coil[0][i]+=center;
}

//calc the line if we can
if(runline)
{
 for(int j=onPt-linect;j<onPt+1;++j){
 if(j>=numpts) break;
 Coil[0][j][0]=ston.x();
 Coil[0][j][1]=ston.y();
 Coil[0][j][2]=ston.z();
 ston+=div;
 }

 curang=endAng; //start angle where we ended
 runline=false; //no more line
 got1=false; //reset the gts flags
 got2=false;
 ToAdd=true; //now we can add the helix part again
}

//stay within 2Pi
curang=fmod(curang+angle+startTH,PI2);
}
std::cout<<"D-shape along "<<axis<<" axis with radius "<<R<<" cm "<<std::endl;
}

//***** MAIN *****
int main(int argc, char **argv)

```

```

{
 MPIworld.start(argc, argv);

 //add our new function to the list
 BiotFunctions.insert("Dcircle", Biot_Dcircle);

 std::string parse="";
 int q=1;
 if(MPIworld.master())
 query_parameter(argc, argv, q++, "input file name:", parse);

 //let ever node know about it
 MPIworld.scatter(parse);

 //creat a Parameter
 Parameters pset(parse);
 pset.addSection("params");
 //the desired MultiCoil section to use
 std::string choose=pset.getParamS("section", "params");

 //get the grid info
 typedef XYZshape<XYZfull> TheGrid;
 pset.addSection("grid");
 Grid<UniformGrid> g1(pset.getParamCoordD("min", "grid"),
 pset.getParamCoordD("max", "grid"),
 pset.getParamCoordI("dim", "grid"));
 TheGrid g2(g1, XYZfull());

 //our master MultiBiot object
 MultiBiot<TheGrid> mycoil(g2,pset, choose);
 mycoil.Controller=MPIworld;

 //calc the field
 mycoil.calculateField();
 //dump out the data
 if(MPIworld.master()){
 mycoil.writeMatlab(pset.getParamS("matout", "params", false, "field.mat"));
 mycoil.write(pset.getParamS("textout", "params", false, "shape.boit"));
 }

 MPIworld.end();
}

```

---

--Special Functions template<class Inter1, Inter2, ..., Inter9>  
--- function  
--- function

---

--- magneticField

--- PreCalc

--- preCalculate

--- TotalMag

--Constructors

--- Interactions

How BlochLib handles the Bloch interactions comprise the bulk of the code in the 'bloch equations' section. The grids, gradients, magnetic fields, and rotations all interact with the interactions to produce highly efficient methods to both assign properties to spins, and to solve the resulting differential equations.

The master class, `Interactions`, acts as the master container for all the interactions in the system. This class is NOTHING without the various interaction classes described below. It provides the main interaction function to interface with the Bloch class to provide the ODE solver with the correct function. It can hold up to 9 different interactions in its template definition.

The interactions can be broken up into three sub categories

- **Particle interactions**...here, no information about the other spins in the system is necessary. Offsets and Relaxation fit into this category.
- **Global interactions**...some total system property(s) is necessary to calculate the interaction. Bulk susceptibility and Radiation Damping fit into this category.
- **Multi Particle interactions**...These interactions are typically the most non-linear of all the interactions. To calculate the necessary information, the current state of the entire spin system must be known, making these interactions also the most time consuming to calculate. Dipoles and the Demagnetizing Field both fit into this category.

Each interaction class has the same set of REQUIRED functions and constants. These functions are designed to interface with the `BlochSolver` and the `Bloch` classes to infer the fastest way to calculate the interactions. To input new interactions of your own, they must have these functions to function in the solver. These functions are listed in the **Special Functions** section and are both valid to the `Interactions` class itself as well as the sub-interaction classes.

Below are the types of interactions and a brief description of each. Functions specific to each interaction are described in their own class section.

### The Particle interactions....

- template<class FieldCalc\_t= NullBFcalc, class Offset\_T=double>  
class Offset

`FieldCalc_t` acts as the Magnetic Field class. The case where `FieldCalc_t` is NOT `NullBFcalc` is described below.

This is the first basic offset class. For `Offset_T =double` (default) and the `FieldCalc_t= NullBFcalc`, the offset is

assumed to be along ONLY the z-axis.

The second case, where `Offset_T =coord<>` is when the magnetic field is NOT along the z-axis, but along an arbitrary axis. Here there are 3 offsets, x,y,z.

- ```
template <class Grid_t >
class
Offset<ListBlochParams<GradientGrid<Grid_t>,
BPos, double >, double>
```

This contains the next basic offset class where a gradient is applied over entire grid, here the offsets can have basic assigned values (the `spin_offset`) and offsets due to gradients. This class will only function for `Offset_T =double` as currently gradients in an off z-axis frame are not yet written.

- ```
template<class FieldCalc_t>
class Offset<FieldCalc_t, FieldCalc_t
::Offset_T >
```

When the `FieldCalc_t` is NOT `NullBFCalc` or `ListBlochParams<GradientGrid<Grid_t>, BPos, double>` it is assumed that the offsets are to be calculated using an input magnetic field class. These classes have either "StaticField" or "DynamicField" as base classes. A `StaticField` base tells the offset class that the field does NOT change in time (thus `PreCalc=0`), but that the `Offset_T=coord<>`. A `DynamicField` base tells the offset will change in time (`PreCalc=1`, how it changes will be up to you to write) and that the `Offset_T=coord<>`.

- ```
template< class Offset_T=double>
class Relax
```

This class simply holds the container for the spin relaxation parameters, T1 and T2. The `Offset_T` should be the same as the `ListBlochParams::Offset_T` and the `Offset::Offset_T`. The T1 and T2 values for each spin are treated simply as exponential decays along the X-Y plane (T2) and the z-axis (T1) for `Offset_T=double` (the analogous planes and main axis for `Offset_T=coord<>`).

The Global interactions....

- `class BulkSus`

The bulk susceptibility relies on the total magnetization of the sample to create a reaction field to the main field, thus causing slight offset change from the basic offset. It is a bulk property, and thus makes no sense for a `BPOptions=BPOptions::Particle`, but only for `BPOptions::Density`.

- `class RadDamp`

Radiation damping is effectively the EMF induced in the coil from the sample polarization back reacting on the sample it self. It is a polarization conserving interaction (not a 'relaxation' mechanism) but behaves like a sort of nonlinear relaxation. It is a bulk property, and thus makes no sense for a `BOptions=BOptions::Particle`, but only for `BOptions::Density`.

The Multi-Particle interactions....

- `template<class Grid_t>`
`class DipoleDipole`

This class maintains the Dipole-Dipole interaction between each point on the grid (not just nearest neighbor, but all neighbors). It determines the dipole coupling through the spins gamma factors and the distance between them. The dipole equation of motion is calculated assuming a HIGH z-axis magnetic field (i.e. truncated).

- `template <class Grid_t>`
`class DimLessDipole`

This class maintains the Dipole-Dipole interaction between each point on the grid (not just nearest neighbor, but all neighbors). It determines the dipole coupling a ratio between some input value (you must specify it) and the distance between the spins such that the largest possible coupling is the one you specify. The dipole equation of motion is calculated assuming a HIGH z-axis magnetic field (i.e. truncated).

- `template<class Grid_t>`
`class DemagField`

This class calculates the demagnetizing field for a given set of spins. This field is related to the asymmetry in a shape, or the non-uniform magnetization of a sample. It is a bulk property, and thus makes no sense for a `BOptions=BOptions::Particle`, but only for `BOptions::Density`. It is designed to be the effective dipolar field that a given sees in a bulk sample and is to replace the 'DipoleDipole' interaction for such density calculations.

- `template<class Grid_t>`
`class DimLessDemagField`

This class calculates the demagnetizing field for a given set of spins. This field is related to the asymmetry in a shape, or the non-uniform magnetization of a sample. It allows you to set the maximal value for

a interaction, rather then relying on the strict distance integration.

Interactions::Special Function

Function: `template<class ParamList>`
`inline void function(double t, Vector<coord> &M, Vector<coord>`
`> &dMdt, ParamList *pars, coord &totM);`

Input: t --> the current time
M --> current Magnetization
dMdt --> the current evaluated differential equation
pars --> the pointer to the Parameter List (this is the ListBlochParams of some kind)
totM --> the current Total Magnetization for the entire spin system

Description: This is the master function for the Interactions class. The Interactions class uses this function to calculate frequency for ALL the spins, along each direction, in the system. The function ADDS the resulting frequencies from the calculation to the appropriate spin in the dMdt vector.

Not all the input parameters are used for every interaction. The bulk susceptibility offset interaction requires the total magnetization (totM), the current magnetization (M) and the list of frequencies (dMdt).

Example

Usage:

Interactions::Special Function

Function: `template<class ParamIter>`
`inline coord<> function(double t, coord<> &M, coord<> &dMdt,`
`ParamIter *pars, coord<> &totM);`

Input: t --> the current time
M --> current Magnetization
dMdt --> the current evaluated differential equation
pars --> the pointer to the Parameter List Iterator (this is the ListBlochParams::iterator of some kind)
totM --> the current Total Magnetization for the entire spin system

Description: This is the master function for the Interactions class. The Interactions class uses this function to calculate frequency for a SINGLE spin, along each direction, in the system. The function returns the resulting frequency 3-vector.

Not all the input parameters are used for every interaction. The basic offset interaction requires only the current magnetization (M) for its calculations.

Example

Usage:

Interactions::Special Function

Function:

```
template<class ParamList>
rmatrix magneticField(double t, Vector<coord> &M, ParamList
*pars, coord &totM);
```

Input:

- t --> the current time
- M --> the current Magnetization
- dMdt --> the current evaluated differential equation
- pars --> the pointer to the Parameter List (this is the ListBlochParams of some kind)
- totM --> the current Total Magnetization for the entire spin system

Description:

- This function returns the magnetic field for each spin in the ParamList in a real square matrix. It calculates the magnetic field as generated by the interaction NOT the frequencies as generated by the ‘function.’ The size of the matrix is the pars.size()*3 x pars.size()*3. The magnetic field along the x-axis (Bx) for the first spin is stored in the (0,0) element of the returned matrix, By is stored in the (1,1) element, Bz is stored in the (2,2) element. The magnetic field along the x-axis (Bx) for the second spin is stored in the (3,3) element of the returned matrix, By is stored in the (4,4) element, Bz is stored in the (5,5) element, and so on for more spins. This output matrix is diagonal.

It takes in the current time (t), the current Magnetization (M), the current evaluated differential equation part (dMdt), the pointer to the Parameter List (pars, this is the ListBlochParams of some kind), and the current Total Magnetization for the entire spin system (totM, this is only calculated if there is an interaction that has TotalMag==1).

For radiation damping, the elements in the returned matrix are given by

```
Bx= 1/(Tr* pars[i].TotMo())*totM.y() ;
By= -1/(Tr* pars[i].TotMo())*totM.x() ;
Bz=0;
```

The evolutionMatrix function returns the valid frequencies for each point.

Example

□

Usage:

Interactions::Special Function

Function: **static const int PreCalc;**

Input:

Description: If PreCalc==1 then before the interaction can be calculated at a give time, several pre-calculations are required to obtain the correct mathematical values. This flag is set to 1 for dipoles and the demagnetizing field where the current magnetic fields at each point is necessary to calculate the interaction. It is also set to one for offsets that have a 'dynamic' field (i.e. time dependant).

Example

Usage:

Interactions::Special Function

Function: `template<class ParamList>`
`void preCalculate(double t, Vector<coord> &M, Vector<coord>`
`> &dMdt, ParamList *pars, coord &totM);`

Input: t --> the current time
M --> current Magnetization
dMdt --> the current evaluated differential equation
pars --> the pointer to the Parameter List (this is the ListBlochParams of some kind)
totM --> the current Total Magnetization for the entire spin system

Description: This function performs the pre-calculation if necessary (it is necessary if PreCalc==1).

The object of this function is to calculate some internal interaction class variable that is time dependant, but not spin dependant. The dipole interaction requires the magnetic field produced by each spin, for a given instant in time, this field is constant and needs to be calculated only once for every spin, thus reducing the calculation time for determination of the dipole interaction in the differential equation. This function calculates the magnetic fields for a given time that is then later used by ‘function’ to calculate the dipolar couplings.

Example
Usage:

Interactions::Special Function

Function: **static const int TotalMag;**

Input:

Description: If TotalMag==1 then this interaction is at least a Global Interaction where the global property (the total magnetization of the spin system) if required to calculate the interaction. This flag is one for radiation damping and bulk susceptibility.

Example

Usage:

Interactions::constructor

Function:

- `template<class Int_1>
Interactions(Int_1 &in)`
- `template<class Int_1, class Int_2>
Interactions(Int_1 &in1, Int_2 &in2)`
- `template<class Int_1, class Int_2, class Int_3>
Interactions(Int_1 &in1, Int_2 &in2, Int_3 &in3)`
- `....`
- `template<class Int_1, class Int_2, ..., class Int_9>
Interactions(Int_1 &in1, Int_2 &in2,..., Int_9 &in9)`

Input:

- `in1..in9` --> the interaction classes

Description:

- The constructor for the Interactions class. It maintains each interaction in as a pointer. The number of template items corresponds to the number of interaction you input. For 5 templates, you only have the 5 interaction constructor. The order of the interaction inputs must be the same as the order of the template arguments.

Example

- `//some typedefs for simplicity in typing`

Usage:

```
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrida; //our shape
typedef RotatingGrid<TheGrida> TheGrid; //spinning grid
typedef ListBlochParams< TheGrid, BOptions::Particle | BOptions::HighField, double > MyPars;

typedef Interactions<Offset<>, Relax<>, BulkSus, RadDamp,
DimLessDipole<TheGrid> > MyInteractions;

//set up the grid
Grid<UniformGrid> gg(coord<>(-1,-1,-1), coord<>(1,1,1),
coord<int>(5,5,5)); //the basic grid
TheShape tester;
TheGrida jja( gg, tester);
TheGrid jj(jja, coord<>(1,1,1), 2000.0*PI2);

//the basic parameters
MyPars mypars(jj.size(), "1H", jj);

//the interaction classes
Offset<> myOffs(mypars, offset);
double t2s=0, t1s=0;
Relax<> myRels(mypars, (!t2s)?0.0:1.0/t2s, (!t1s)?0.0:1.0/t1s);
BulkSus BsRun(1.0);
RadDamp RdRun(0.01);
DimLessDipole<TheGrid> DipDip(jj, 3230.0*PI2);
//turn on the dynamic flag as spinning the grid
DipDip.Dynamic=true;

//a 5 member interaction class
```

```
MyInteractions MyInts(myOffs, myRels,BsRun, RdRun, DipDip);
```

--Constructors template<class FieldCalc_t=NullBFcalc, class Offset_T=typename FieldCalc_t::Offset_T>
--- **Offset** class Offset;

--- **Offset**

--- **Offset**

--- **Offset**

--- **Offset**

--Element Extraction

--- **Bo**

--- **offset**

--- **spinOffset**

--Assignments

--- **Bo**

--- **offset**

--- **spinOffset**

--Other Functions

--- **gradOn**, **gradOff**

--- **off**

--- **on**

FieldCalc_t acts as the 'Magnetic Field' class. It can be one of three class types

- **NullBFcalc**...this class is a 'NULL' magnetic field calculator. Meaning that there is NO built in way to calculate the offsets. You will need to specify each offset for each spin. The offsets here to not change in time.

The master function for the i'th spin returns

```
wx= offset(i)*M[i].y()  
wy= -offset(i)*M[i].x()  
wz= 0
```

The **Offset_T** can be set to **coord<>**. This implies that the magnetic field is not only along the z-axis, and that the **ListBlochParams** has an offset type =**coord<>** as well, if not the program will not compile. The simplest way to setup such a offset interaction is via this syntax

```
Offset<ListBlochParams<Grid_t ,  
BOptions, coord<> > myOffs;
```

The master function for the i'th spin returns

```
wx= cross(offset(i),M[i]).y()  
wy= cross(offset(i),M[i]).x()  
wz= cross(offset(i),M[i]).z()
```

- **ListBlochParams<GradientGrid<GridEngine_t>>...**
... This is a gradient. Here the offsets can be have both a 'spin_offset' meaning a nominal value for the offset, AND an offset due to the gradient applied. The total offset seen by the integrator is then the sum of the two offsets.

The master function for the i'th spin returns

```
wx= (spin_offset(i)+dot(G,  
Point(i)))*M[i].y()  
wy= -(spin_offset(i)+dot(G,  
Point(i)))*M[i].x()  
wz= 0
```

No offset interaction is defined for **Offset_T=coord<>**, and the interaction will be invalid.

- **BCalculator**...this can be some arbitrary class that is either a subclass of 'StaticField' or 'DynamicField.' If it is not, then you will get a compilation error. The magnetic fields should be in Gauss. The BCalculator class also REQUIRES these functions

```
Offset_T Bfield(int i)
Offset_T Bfield(double t, int i)
```

that returns the magnetic field at point i and at time t, EVEN IF the field is static. If the Offset_T=double, then the field is assumed to be along the z-axis (as in the NullBFcalc case). The offset at point i and at time t is assigned

```
offset(i)=( BCalculator::Bfield(t, i)
-Bo)*ListBlochPars<...>::gammaGauss(i);
```

where 'Bo' is some global magnetic field (it is the same type as Offset_T). The default for Bo is 0. If Offset_T = double then the master function for the i'th spin returns

```
wx= offset(i)*M[i].y()
wy= -offset(i)*M[i].x()
wz= 0
```

If Offset_T =coord<> then the master function for the i'th spin returns

```
wx= cross(offset(i),M[i]).y()
wy= cross(offset(i),M[i]).x()
wz= cross(offset(i),M[i]).z()
```

Offset::constructor

Function:

□ **Offset()**

Input:

□ void

Description:

□ The empty constructor.

sets the interation on, and all offsets to 0

Example Usage:

□ `Offset<> myOff;`

Offset::constructor

Function: `template<class ...>
Offset(ListBlochParams< ... > &bc, Offset_T offset=0.0);`

- Input:**
 - bc--> the ListBlochParams sets offset's size.
 - offset--> the offset of type 'Offset_T' that will set the entire list to this number. If using a 'BCcalculator', then this variable it the global magnetic field (the rotating frame)
- Description:**
 - Uses the input ListBlochParams to both set the size of the offset data vector, and if nessesary, be used to calculate the offsets if using a magnetic field. If using the BCcalculator, the 'offset' input is accutually the 'Bo' value (the global magnetic field, the rotating frame for the system).

Example

```
//create a grid  
Grid<UniformGrid> gg(mins, maxs, dims);  
  
//create a shape  
XYZshape<XYZfull> myShape(gg, XYZfull());  
  
//create the gradient  
GradientGrid<XYZshape<XYZfull> > myGrad(myShape);  
  
//create the List Bloch Params with a gradient  
typedef ListBlochParams< GradientGrid<XYZshape<XYZfull>  
>,BPOptions::HighField | BPOptions::Particle, double> myGradList;  
  
int numSpins=myGrad.size();  
std::string spinType="1H";  
MyPars myList(numSpins,spinType, myGrad);  
  
//this is the basic SpinOffset for all the  
// spins in the list  
//IT IS IN RADS/SEC  
double basicOff=200*PI*2;  
  
//create an offset using the gradient  
Offset<MyPars> myOff(myPars, basicOff);  
  
//create a basic Offset where all offsets  
// is set to basicOff  
Offset<> myOff2(myPars, basicOff);
```

Offset::constructor

Function: `Offset(int &len, Offset_T offset=0.0):`

Input:

- len --> the number of spins in the offset object
- offset --> the offset value given to all the spins in RAD/SEC

Description:

- A constructor for the NullBFcalc object. It sets the size of the object (should be the number of spins) and set their offsets to the same value, 'offset.'

Example

- `//an offset double type with 50 spins`

Usage:

- `// with an offset of 300 Hz`

```
Offset<> myOff(50, 300.0*PI*2);
```

```
//an offset coord<> type with 50 spins
```

```
// with an offset of (0,0,300 Hz)
```

```
Offset<NullBFcalc, coord<> > myOff2(50, coord<>(0,0,300*PI*2);
```

Offset::constructor

Function: `template<class GridEngine_t, int BPop>`
`Offset(BCalculator &bc,`
`ListBlochParams<GridEngine_t, BPop, double> &lp,`
`Offset_T Bo=0):`

Input: bc--> a Magnetic Field class with valid functions (Offset_T Bfield(double t, int i) and Offset_T Bfield(int i). Ths object should be public of 'StaticField' or 'DynamicField' for the code to compile.
lp--> the lis of Bloch Parameters. This is nessesary to obtain the correct gamma factor for each spin.
Bo--> the 'rotating frame' of the system (in Gauss)

Description: This is the basic constructor for the BCalculator type of Offset. It maintains a pointer to the BCalculator object (thus if you destroy it before destroying the Offset, the Offset object will likely crash the program). It initializes all the magnetic fields of the ListBlochParams to the element in the list given by the BCalculator's functions Bfield(double t, int i) and Bfield(int i). If Bo is present all the offsets will be calculated by subtracting this 'rotating frame' field from the fields given by the BCalculator's functions.

Example `//set up a static field class`

Usage: `class myRandomField:`
`public StaticField`
`{`
`Random<> rand;`
`myRandomField():`
`rand(0.0, 100.0)`
`{};`
`double Bfield(double t, int i)`
`{ return Bfield(i); }`

`double Bfield(int i)`
`{`
`return rand(); //a random field from 0..100 Gauss`
`}`
`};`

`//declare a BCalculator object`
`myRandomField myF;`

`//set up the grids`

`//create a grid`
`Grid<UniformGrid> gg(mins, maxs, dims);`

`//create a shape`
`XYZshape<XYZfull> myShape(gg, XYZfull());`

`//create the List Bloch Params with a gradient`
`typedef ListBlochParams< XYZshape<XYZfull>, BPOptions::HighField |`
`BPOptions::Particle, double> myList;`

`int numSpins=myGrad.size();`
`std::string spinType="1H";`

```
MyPars myList(numSpins,spinType, myShape);  
  
//declare the offset object  
//using the rotating frame of BO=50 Gauss  
// (half the random range in the above class)  
Offset<myRandomField, double> myOffs(myF, myList, 50.0*PI2)
```

Offset::constructor

Function: `Offset(const Vector<Offset_T> &offset):`

Input: `offsets-->` a Vector of the Offset_T type that define each of the spins' offset in RAD/SEC.

Description: `Constructor for a NullBFcalc offset type. The offsets get set to the input data vector. The sizew of the object is also the size of the input vector.`

Example `//set up a offset vector`

Usage: `// from 0..50Hz
Vector<double> offV(Spread<double>(0., 50*PI2, 2*PI2));
Offset<> myOffs(offV);`

Offset::element extraction

Function: `Offset_T &Bo()`

Input: `void`

Description: `Returns the 'rotating frame' magnetic field (in Gauss) for the BCalculator offset types.`

Example `//set up a static field class`

Usage: `class myRandomField:`

```
public StaticField
{
    Random<> rand;
    myRandomField():
        rand(0.0, 100.0)
    {};
    double Bfield(double t, int i)
    { return Bfield(i); }

    double Bfield(int i)
    {
        return rand(); //a random field from 0..100 Gauss
    }
};

//declare a BCalculator object
myRandomField myF;

//set up the grids

//create a grid
Grid<UniformGrid> gg(mins, maxs, dims);

//create a shape
XYZshape<XYZfull> myShape(gg, XYZfull());

//create the List BLoch Params with a gradient
typedef ListBlochParams< XYZshape<XYZfull>, BOptions::HighField | BOptions::Particle, double> myGradList;

int numSpins=myGrad.size();
std::string spinType="1H";
MyPars myList(numSpins,spinType, myShape);

//declare the offset object
//using the rotating frame of BO=50 Gauss
// (half the random range in the above class)
Offset<myRandomField, double> myOffs(myF, myList, 50.0*PI2)

//prints 50*PI2
cout<<myOffs.Bo()<<endl;
```

Offset::element extraction

Function: `Offset_T offset(int i)`

Input: `i-->` the spin index in the list

Description: `>Returns the offset at element i in Rad/sec.`

if the Offset type is with a Gradient Grid then this returns the FULL offset

```
spinOffset(i)*dot(G,Point(i))*gammaGauss(i))
```

Example Usage: `//an offset double type with 50 spins`

```
// with an offset of 300 Hz
```

```
Offset<> myOff(50, 300.0*PI*2);
```

```
cout<<myOff.offset(3); //will return 300*PI2
```

Offset::element extraction

Function: `Offset_T spinOffset(int i)`

Input: `i-->` the spin index

Description: `>Returns the spin offset for a given Gradient based Offset object. It returns the spin offset NOT including the gradient.`

Example `//create a grid`

Usage: `Grid<UniformGrid> gg(mins, maxs, dims);`

```
//create a shape
XYZshape<XYZfull> myShape(gg, XYZfull());

//create the gradient
GradientGrid<XYZshape<XYZfull> > myGrad(myShape);

//create the List Bloch Params with a gradient
typedef ListBlochParams< GradientGrid<XYZshape<XYZfull>
>,BOptions::HighField | BOptions::Particle, double> myGradList;

int numSpins=myGrad.size();
std::string spinType="1H";
MyPars myList(numSpins,spinType, myGrad);

//this is the basic SpinOffset for all the
// spins in the list
//IT IS IN RADS/SEC
double basicOff=200*PI*2;

//create an offset using the gradient
Offset<MyPars> myOff(myPars, basicOff);

//will print 300*PI2*dot(G, Point())
cout<<myOff.offset(5)<<endl;

//will print 300*PI2
cout<<myOff.spinOffset(5)<<endl;
```

Offset::assignments

Function: `void Bo(Offset_T newBo)`

Input: `newBo`--> the new rotating fram magnetic field (in Gauss) of the Offset_T type

Description: `Sets the 'rotating frame' magnetic field (in Gauss) for the BCalculator offset types.`

Example `//set up a static field class`

Usage: `class myRandomField:`

```
public StaticField
{
    Random<> rand;
    myRandomField():
        rand(0.0, 100.0)
    {};
    double Bfield(double t, int i)
    { return Bfield(i); }

    double Bfield(int i)
    {
        return rand(); //a random field from 0..100 Gauss
    }
};

//declare a BCalculator object
myRandomField myF;

//set up the grids

//create a grid
Grid<UniformGrid> gg(mins, maxs, dims);

//create a shape
XYZshape<XYZfull> myShape(gg, XYZfull());

//create the List BLoch Params with a gradient
typedef ListBlochParams< XYZshape<XYZfull>, BOptions::HighField | BOptions::Particle, double> myList;

int numSpins=myGrad.size();
std::string spinType="1H";
MyPars myList(numSpins,spinType, myShape);

//declare the offset object
//using the rotating frame of BO=50 Gauss
// (half the random range in the above class)
Offset<myRandomField, double> myOffs(myF, myList, 50.0*PI2)

//set the new rotating field
myOff.Bo(60*PI2);
```

Offset::assignments

Function: `double &offset(int i)`

Input: `i-->` the spin index

Description: This sets the offset for the spin index `i` (in RAD/SEC). (Assignment will NOT work for gradient grid offset types). You can use this function for the BCalculator Offset types, but note that if the BCalculator is a subclass of 'DynamicField' then it will be calculated based on the field, not the value you specify.

Example `//set up a offset vector`

Usage: `// from 0..50Hz`
`Vector<double> offV(Spread<double>(0., 50*PI2, 2*PI2));`
`Offset<> myOffs(offV);`

`//set the offset for the 5'th spin`
`myOffs.offset(4)=500*PI2;`

Offset::assignments

Function: `double &spinOffset(int i)`

Input: `i-->` the spin index

Description: This sets the offset for the spin index `i` (in RAD/SEC). (Assignment will work for gradient grid offset types). You can use this function for the BCalculator Offset types, but note that if the BCalculator is a subclass of 'DynamicField' then it will be calculated based on the field, not the value you specify.

Example `//set up a offset vector`

Usage: `// from 0..50Hz`
`Vector<double> offV(Spread<double>(0., 50*PI2, 2*PI2));`
`Offset<> myOffs(offV);`

`//set the offset for the 5'th spin`
`myOffs.spinOffset(4)=500*PI2;`

Offset::other

Function: **void gradOn();**
 void gradOff();

Input: void

Description: Will turn the gradient on (gradOn) or off (gradOff) IF the offset type is a Gradient type. If the interaction is off, then the total offset is just the spinOffset (to turn everything off you must use the 'off()' function).

Example `//create a grid`

Usage: `Grid<UniformGrid> gg(mins, maxs, dims);`

```
//create a shape
XYZshape<XYZfull> myShape(gg, XYZfull());

//create the gradient
GradientGrid<XYZshape<XYZfull> > myGrad(myShape);

//create the List Bloch Params with a gradient
typedef ListBlochParams< GradientGrid<XYZshape<XYZfull>
>,BPOptions::HighField | BPOptions::Particle, double> myGradList;

int numSpins=myGrad.size();
std::string spinType="1H";
MyPars myList(numSpins,spinType, myGrad);

//this is the basic SpinOffset for all the
// spins in the list
//IT IS IN RADS/SEC
double basicOff=200*PI*2;

//create an offset using the gradient
Offset<MyPars> myOff(myPars, basicOff);

//turn the Gradient Off
myOff.gradOff();
```

Offset::other

Function:

□ **void off()**

Input:

□ void

Description:

□ This will turn the interaction off.

Example Usage:

```
□ Offset<> myOff(4, 300*PI2);
  //turn off all the offsets
  myOff.off();

  //turn them back on
  myOff.on();
```

Offset::other

Function:

□ **void on()**

Input:

□ **void**

Description:

□ This will turn the interaction on (this is the default behavior).

Example Usage:

```
□ Offset<> myOff(4, 300*PI2);
  //turn off all the offsets
myOff.off();

//turn them back on
myOff.on();
```

--Constructors

```
template<class Offset_T=double>
class Relax { ... }
```

This is the Relaxation interaction. Relaxation is treated quite simply ad involves 2 parameters, a T2 (or dephasing or transverse or reversible) relaxation and a T1 (or longitudinal or non-reversible).

--Element Extraction

--- Mo

--- T1, T2

--- T2, T1

--Assignments

--- setMo

--- T1, T2

--- T2, T1

Examples

--- coord Relax

wx=-Mx*T2

wy=-My*T2

T2 is typically thought of as a phase relaxation, it does not decrease the magnetic contribution of an individual spin, but as a bulk, the magnetic vectors become dephased with respect to one another. Thus the total magnetization appears decreased when summed up. Things like pulse imperfection, dipole-dipole, and field inhomogeneities cause this relaxation and theoretically they can all be refocused (reversed) using proper pulse sequences. The interaction here is treated on a spin-by-spin basis where each 'spin' is considered a 'density' of spins, and this little density is dephased. Thus this interaction makes little sence for a BOptions::Particle (but it is not disallowed). Mathematically it effects the M_x and M_y components of the magnetization and effectively shrinks them at a rate of 1/T2.

wz=(Mo.z() - M.z()) * T1

If the main magnetic axis is NOT along the z-axis, but along some axis, n=(n₁, n₂, n₃), (the Offset_T=coord<> case), then T2 effects the plane perpendicular to this axis and T1 attempts to move the magnetization back to the n axis.

Both T2 and T1 are still scalar interactions even in a tilted frame. Assuming that you have set the n axis (and that is does not change in time), then we need to rotate our x,y,z axis set into this new frame. Thus we need some Euler angles which can be determined from the given n-axis. We can only determine theta and phi from the axis and not gamma, as we do not know the rotation of that tilted frame. However, the transverse relaxation spans that entire plane, so gamma is not really necessary.

```
theta=acos(n_z/norm(n))
phi=atan(n_x/n_y)
```

we can then rotate our interaction back our normal basis by doing the inverse rotation, these equations can be written in our Cartesian basis as

```
w_x=-
T2*cos(phi)*M_x+sin(phi)*(T2*cos(theta)*M_y+T1*sin(theta)*(Mo-
M_z)) ,
w_y=-T2*cos(phi)*cos(theta)*M_y-
T2*M_x*sin(phi)+T1*cos(phi)*sin(theta)*(M_z-Mo) ,
w_z=T1*cos(theta)*(Mo-M_z)-T2*M_y*sin(theta)
```

where Mo is the magnitude of the n of the axis. The only way we could have a relaxation axis different from the z-axis is if our magnetic field axis was off of the z-axis. Thus this sort of interaction only makes sence for both Offset_T of the ListBlochParams and the Offset_T of the Offset to be coord<>.

Currently there is NO relaxation that would have a time dependant theta and phi (theta=theta(t) and phi=phi(t)) this is simply because the parameters 'T1 and T2' should also fluctuate under such a time dependant Hamiltonian. Modeling such a system would require some Redfield theory and is out of the scope of this library.

Relax::constructor

Function:	<input type="checkbox"/> Relax()
Input:	<input type="checkbox"/> void
Description:	<input type="checkbox"/> The empty constructor. Sets the interaction size to 0.
Example Usage:	<input type="checkbox"/> <pre>//an empty Offset_T=double relaxation class Relax<> mRel;</pre> <input type="checkbox"/> <pre>//an empty Offset_T=coord<> relaxation class Relax<coord<> > myRel2;</pre>

Relax::constructor

Function: `Relax(int len, double inT2=0, double inT1=0)`

Input:

- len--> the length of the interaction (should be the size of a ListBlochParams object)
- inT2--> the T2 relaxation parameter (dephasing) in 1/SECONDS (If this is 0, then, it is assumed that there is NO T2 interaction)
- inT1--> the T1 relaxation parameter (transverse) in 1/SECONDS (If this is 0, then, it is assumed that there is NO T1 interaction)

Description: □ Sets the size of the object to 'len' with the values inT2 and inT1 for every spin in the list.

Example `//a relaxation object of size 5, and t2=0.02 sec`

Usage: `Relax<> myRel(5, 0.02);`

```
//a relaxation object of size 5, t2=0.02 sec, and t1=0.01 sec
Relax<> myRel2(5, 0.02, 0.01);
```

```
//a coord<> version
Relax<coord<> > myRel3(5, 0.02, 0.01);
```

Relax::constructor

Function: `Relax(const Vector<double> &inT2,const Vector<double> &inT1)`

Input:

- inT2--> a vector of T2 relaxation parameters (dephasing) in 1/SECONDS for each spin (If this is 0, then, it is assumed that there is NO T2 interaction)
- inT1--> a vector of T1 relaxation parameters (transverse) in 1/SECONDS for each spin(If this is 0, then, it is assumed that there is NO T1 interaction)

Description:

- Creates a Relaxation object using the input vectors for T2 and T1. Both vectors should be the SAME size.

Example `//create 2 vector spreads`

```
//T2...0...6 seconds
Vector<double> T2s(Spread<double>(0,6, 2));
//T1...6...0 seconds
Vector<double> T1s(Spread<double>(6, 0, -2));
```

```
//a realxation object
```

```
Relax<> myRel(T2s, T1s);
```

```
//the coord<> version
```

```
Relax<coord<> > myRel2(T2s, T1s);
```

Relax::constructor

Function:

- `template<class GridEngine_t, int BPos, class Offset_T>`
`Relax(ListBlochParams<GridEngine_t, BPos, Offset_T> &bc, double`
`inT2=0, double inT1=0)`

Input:

- bc--> a ListBlochParams object that sets the size of the object.
- inT2--> the T2 relaxation parameter (dephasing) in 1/SECONDS (If this is 0, then, it is assumed that there is NO T2 interaction)
- inT1--> the T1 relaxation parameter (transverse) in 1/SECONDS (If this is 0, then, it is assumed that there is NO T1 interaction)

Description:

- Sets the size of the object to length of the ListBlochParams with the values inT2 and inT1 for every spin in the list. If the Offset_T=coord<> in Relax, then it will set the Relaxation interaction axis to the axis where Mo lies (as well as obtain the magnitude of Mo). So if you have not set the Mo or Bo in the ListBlochParams, then it will default to (0,0,Mo). You can use the 'setMo(ListBlochParam)' to reset this axis from an already declare Relax<coord<> object.

Example

- `//set up the grids`

Usage:

```
//create a grid
Grid<UniformGrid> gg(mins, maxs, dims);

//create a shape
XYZshape<XYZfull> myShape(gg, XYZfull());

//create the List Bloch Params with a gradient
typedef ListBlochParams< XYZshape<XYZfull>, BOptions::HighField | 
BOptions::Particle, double> myList;

int numSpins=myGrad.size();
std::string spinType="1H";
MyPars myList(numSpins,spinType, myShape);

//set up a relaxation object
Relax<> myRel(myList, 0.4, 0.3)
```

Relax::element extraction

Function: `Vector<coord<> > Mo();`
`coord<> Mo(int i);`

Input: `i-->` the spin index you wish to obtain

Description: This is valid for the 'coord<>' version where this vector is the direction where the longitudinal relaxation relaxes to. The transverse relaxation relaxes the components in the plane perpendicular to this axis. It does not check the vector bounds (for speed) so accessing an element greater then the vector size will likely cause the program to crash.

Example `//set up an object`

Usage: `Relax<coord<> > myRel(5, 0.02, 0.01);`

```
//the default rhat vector is (0,0,1) (z-axis)
// for all spins...
cout<<myRel.Mo();

//get the second spins
cout<<myRel.rhat(1); //prints [0 0 1]
```

Relax::element extraction

Function: `double T1(int i);`
`double T2(int i);`

Input: `i--> the spin index`

Description: Returns the individual T1 (or T2) for the spin at index i. It does NOT check the bounds (so imputting an out of bounds index will more then likely crash the program).

Example `//create 2 vector spreads`

Usage: `//T2...0...6 seconds`
`Vector<double> T2s(Spread<double>(0,6, 2));`
`//T1...6...0 seconds`
`Vector<double> T1s(Spread<double>(6, 0, -2));`

```
//a relaxation object
Relax<> myRel(T2s, T1s);

//get a single spin T2 value
cout<<myRel.T2(0); //prints '0'
//get a T1 value
cout<<myRel.T1(1); //prints '4'
```

Relax::element extraction

Function:

```
□ Vector<double> T2();  
□ Vector<double> T1();
```

Input:

```
□ void
```

Description:

```
□ Returns the T2 (or the T1) data vector in the object.
```

Example Usage:

```
□ //set up an object  
Relax<> myRel(50, 0.02);  
  
//the entire 50 spin vector with each element=0.02  
Vector<double> myRel.T2();  
  
//the entire 50 spin vector with each element=0  
Vector<double> myRel.T1();
```

Relax::assignments

Function:

- `void setMo(const Vector<coord> &newMo);`
- `void setMo(int i,const coord& &newMo);`
- `void setMo(const coord& &newMo);`

- `template<class GridEngine_t, int BPop, class Offset_T>`
- `void setMo(ListBlochParams<GridEngine_t, BPop, Offset_T > &Lbc);`

Input:

- `newMo`--> A Vector of `coord<>` (or a single `coord<>` for spin *i*) that sets the new relaxation axis.
- `Lbc`--> a `ListBlochParams` with the `Mo` set to the coorect axis (if `Offset_T=double`, the axis always be $(0,0,1)$)

Description:

- This is valid for the '`coord<>`' version where this vector is the direction where the longitudinal relaxation relaxes to. The transverse relaxation relaxes the components in the plane perpendicular to this axis. This either sets the direction for a single spin (at index *i*) or the enitre object's `rhat` vector. If using the '`setMo(coord<>)`' version then ALL the spins get set to this axis. It also normalizes the input `coord<>` (even if you did not). Using '`setMo(ListBlochParams)`' will set interaction axis to the one given in by the `Mo` of the `ListBlochParams`.

Example

```
//set up an object
Relax<coord> > myRel(5, 0.02, 0.01);

//the default rhat vector is (0,0,1) (z-axis)
//set all the spins to relax to the x axis
myRel.setMo(coord<>(14,0,0));

// set the 2nd spin with a (1,0,1) axis
myRel.setMo(2,coord<>(1,0,1));
```

Usage:

Relax::assignments

Function: `double &T1(int i);`
`double &T2(int i);`

Input: `i-->` the spin index

Description: Returns the individual reference to the T1 (or T2) for the spin at index i. It does NOT check the bounds (so inputting an out of bounds index will more then likely crash the program).

Example `//a relaxation object of length 3`

Usage: `Relax<> myRel(3);`

```
//get a single spin T2 value
myRel.T2(0)=0.3; //set the first spins T2 value to 0.3
myRel.T1(1)=2; //set the second spins T1 value to 2
```

Relax::assignments

Function: `Vector<double> &T2();`
 `Vector<double> &T1();`

Input: `void`

Description: Returns the reference to the vector data object so that you can set a new data vector.

Example Usage: `//create 2 vector spreads`
`//T2...0...6 seconds`
`Vector<double> T2s(Spread<double>(0,6, 2));`
`//T1...6...0 seconds`
`Vector<double> T1s(Spread<double>(6, 0, -2));`

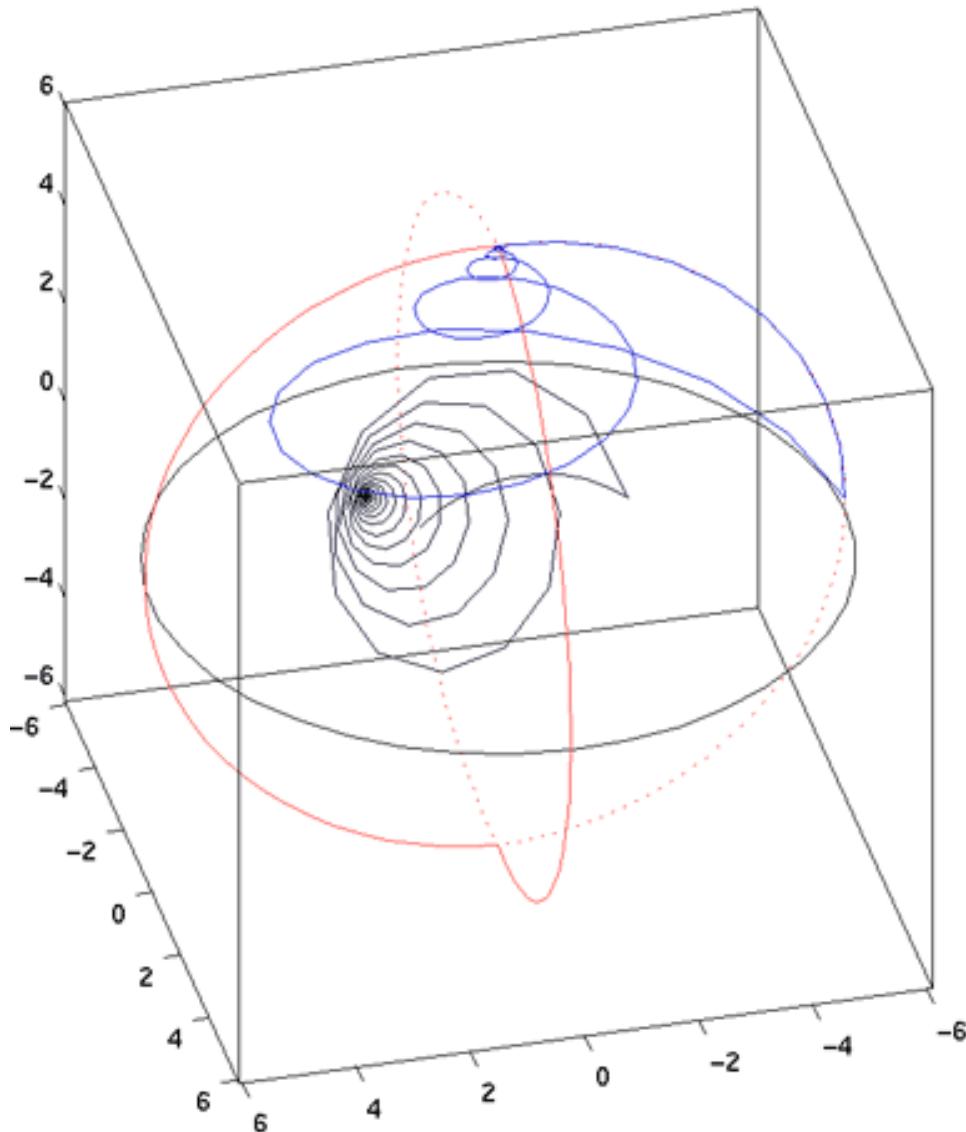
`//set up an empty object`
`Relax<> myRel;`

`//set the T2 vector`
`myRel.T2()=T2s;`

`//set the T1 vector`
`myRel.T1()=T1s;`

using the coord<> in Relax

This is a simple 90 pulse on a 2 spin system, where there magnetic field for each can be set to a different 3-vector. Thus the offsets are 3 vectors, and the relaxation parameters are performed on a shifted axis. Below is the input file and the code for such a simple system. Only 2 Bo and 2 offset are allowed (but n spins are allowed for the grid). Using the **plottrag** matlab function, the below trajectories for the 2 spins are plotted.



A plot of the spin magnetization trajectory. The gray line corresponds to the spins aligned along the $\text{Bo} = (1, 1, 1)$ axis, the blue trajectory are spins aligned along the $\text{Bo} = (0, 0, 1)$ axis.

-----The Input File-----

```
#parameter file for 2 pulse sequences

dim 1,1,2
min -0.5,-0.5,-0.5
max 0.5, 0.5, 0.5

#fid pieces
npts 1024
tf 0.5

#the pulse bits
pulseangle 90
pulsephase 270
pulseamp 80000

#basic spin parameters
Bo1 3,3,3
Bo2 0,0,10

temperature 300
offset1 100,100,100
offset2 0,0,100
T2 0.01
T1 0.01
spintype 1H
moles .2

#calculate lypuvnovs[1] or not[0]
lyps 0
lypout lyps

#file output names for the data
fidout data
magout mag
trajectories traj
```

-----End Input File-----

```
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

timer stopwatch;
void printTime(int nrounds=1){
    std::cout << std::endl << "Time taken: "
    << (stopwatch() / nrounds) << " seconds
";
}

void Info(std::string mess)
{
```

```

cout<<mess<<endl;
cout.flush();
}

int main(int argc,char* argv[])
{
    std::string fn;

//the parameter file
    query_parameter(argc,argv,1, "Enter file to parse: ", fn);
    Parameters pset(fn);

//get the basic parameters
    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");
    double inTemp=pset.getParamD("temperature");
    string spintype=pset.getParamS("spintype");
    string detsp=spintype;
    double t2s=pset.getParamD("T2");
    double t1s=pset.getParamD("T1");
    double moles=pset.getParamD("moles");

    std::string fout=pset.getParamS("fidout");
    std::string magout=pset.getParamS("magout");

    coord<int> dims(pset.getParamCoordI("dim"));
    coord<> mins(pset.getParamCoordD("min"));
    coord<> maxs(pset.getParamCoordD("max"));

    int cv=pset.getParamI("lyps");

    std::string dataou=pset.getParamS("trajectories", "", false);

// Grid Set up
    typedef XYZfull TheShape;
    typedef XYZshape<TheShape> TheGrid;

    Info("Creating grid....");
    Grid<UniformGrid> gg(mins, maxs, dims);

    Info("Creating initial shape....");
    TheShape tester;
    Info("Creating total shape-grid....");
    TheGrid jj( gg, tester);

//List BlochParameters
    typedef ListBlochParams<
        TheGrid,
        BOptions::Density | BOptions::HighField,
        coord<> > MyPars;
    int nsp=jj.size();
    Info("Creating entire spin parameter list for "+itost(nsp)+" spins....");
    MyPars mypars(nsp, "1H", jj);
    nsp=mypars.size();
}

```

```

//The pulse list for a real pulse on protons..
Info("Creating real pulse lists...");

//get the info from the pset
double pang=pset.getParamD("pulseangle");
double amp=pset.getParamD("pulseamp");
double phase=pset.getParamD("pulsephase");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype, amp*PI2, phase*DEG2RAD);

//get the first Bo
coord<> inBol=pset.getParamCoordD("Bo1");
//get the second Bo
coord<> inBo2=pset.getParamCoordD("Bo2");

Info("Setting spin parameter offsets....");
for(int j=0;j<nsp;j++){
    mypars(j)=spintype;
    mypars(j).moles(moles);
//set the proper Bo
    if(j%2==0) mypars(j).Bo(inBol);
    else mypars(j).Bo(inBo2);
    mypars.temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(cout);
PP1.print(cout);

//get the time for the 90 pulse
double tpulse=PP1.timeForAngle(pang*Pi/180., spintype);

//the time trains
Info("Initializing Time train for first Pulse....");
TimeTrain<UniformTimeEngine>
    P1(UniformTimeEngine(0., tpulse, 10,100));

Info("Initializing Time train for FID....");
TimeTrain<UniformTimeEngine>
    F1(UniformTimeEngine(tpulse, tpulse+tf, nsteps,20));

//Extra interactions
typedef Interactions<
    Offset<NullBFcalc,
    coord<> >,
    Relax<coord<> > > MyInteractions;
Info("Setting Interactions....");

//the offsets
Offset<NullBFcalc, coord<> > myOffs(mypars);
//get the first offset
coord<> offset1=pset.getParamCoordD("offset1")*PI2;
//get the seconf offset
coord<> offset2=pset.getParamCoordD("offset2")*PI2;

```

```

for(int i=0;i<mypars.size();++i){
    if(i%2==0) myOffs.offset(i)=offset1;
    else myOffs.offset(i)=offset2;
}

//Relaxation off axis
// using the Bo as the main Mo axis....
Relax<coord> > myRels(mypars, (!t2s)?0.0:1.0/t2s, (!t1s)?0.0:1.0/t1s);

//total interaction obect
MyInteractions MyInts(myOffs, myRels);

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//THis is the BLoch solve to perform a pulse
Info("Initializing total parameter list with a pulse....");
PulseBloch myparspulse(mypars, P1, MyInts);
if(cv) myparspulse.calcVariational();

//This is the Bloch solver to Collect the FID (i.e. has no pulses...FASTER)
Info("Initializing total parameter list for FID collection....");
NoPulseBloch me;
me=(myparspulse);
Info("Integrating first Pulse....");

//out initial condition
Vector<coord> > tm=me.currentMag();

stopwatch.reset();
BlochSolver<PulseBloch > drivP(myparspulse, tm, "out");

//output trajectory data if wanted
std::ofstream trajout;
if(dataou!=""){
    trajout.open(dataou.c_str());
    drivP.setCollectionPolicy(All);
}else{
    drivP.setCollectionPolicy(FinalPoint);
}

//integrate the Pulse
drivP.setWritePolicy(Hold);
if(!drivP.solve(P1)){
    Info(" ERROR!!..could not integrate pulse P1....");
    return -1;
}else if(dataou!=""){
    drivP.writeData(trajout);
}

//the fids initial condition is just the previous
// integrations last point
BlochSolver<NoPulseBloch > driv(me, drivP.lastPoint());

Info("Integrating FID ....");

```

```
//set various data collection policies
    std::string lypname="lyps";
    if(dataou!=""){
        driv.setCollectionPolicy(All);
    }else{
        drivP.setCollectionPolicy(MagAndFID);
    }
    driv.setWritePolicy(Hold);

//set the detection spin
    driv.setDetect(detsp);
    if(cv) {
        driv.setLyapunovPolicy(LypContinous);
        driv.setLypDataFile(lypname);
    }

//integrate the FID
    if(driv.solve(F1)){
        //dump out the data
        driv.writeSpectrum(fout);
        driv.writeMag(magout);
        if(dataou!="") driv.writeData(trajout);
    }

    printTime();
}
```

--Constructors[--- BulkSus](#)[--- BulkSus](#)**--Element Extraction**[--- D](#)**--Assignments**[--- D](#)**Examples**[--- the effect](#)

class BulkSus{...

Bulk susceptibility is a high field effect. Classically it is an objects 'reaction' to an applied field. This reaction gives the object its own field that when combined with the global field, produces a new field usually different from the global field. If M is the magnetization of the sample, B the global field, the total field in the sample, H , is

$$H = B + D * M$$

where D is the susceptibility factor of the sample. D is a scalar number from 0..1 that involves a complex integral over the spatial distribution of the sample. For completely symmetric samples, (i.e. a sphere) $D=0$, for a flat disk $D=1/3$, and for a long cylinder $D=1$. A liquid NMR tube can be thought of as a long cylinder. For highly magnetized NMR samples (like water), this bulk susceptibility is significant.

In real practice, a more complex interaction is actually taking place. Local fields from each spin contribute to the observed magnetic fluctuation. This is the demagnetizing field, but the Bulk susceptibility is a good first approximation to general behavior of highly magnetized samples. It should be noted that the magnetization ($\sim 10e-5$ Gauss or less) of a sample is very small when compared to the main B_0 field ($\sim 10,000$ Gauss). So the effects along the B_0 direction are all that really enter into the equations of motion, as the other directions are truncated. Because the field is the only thing effected by this interaction, the offsets are susceptible to this effect.

If our samples magnetization was static, then the bulk susceptibility would only add a small offset to the spins, and would not be noticeable (a simple rotating frame transformation would correct for this effect), however, in NMR we have the ability to manipulate the magnetization directly with RF pulses. Moving the magnetization away perpendicular to the B_0 axis effectively removes bulk susceptibility effect in high fields, however, relaxation or dipole-dipole interactions will eventually replace some of magnetization along the B_0 axis, and then the interaction will begin to appear. It typically appears in spectra as an offset 'chirping' where initially the offset starts at a small value and migrates to higher values as the sample relaxes (the M_z component gets larger and larger, thus H gets larger, and the offset increases). If we include pulse gradients or any other complex manipulations, then the effect of the bulk susceptibility becomes more difficult to describe leaving us to simulate 'what happens.'

This interaction requires the total magnetization of the sample to be calculated correctly. If M_t is our total magnetization, and B_0 is along the z-axis, then our interaction is calculated in the B_0 rotating frame as

$$H = D * M_t z$$

thus adding, to our frequencies

$$\begin{aligned}wx &= D * \text{gamma} * M_{t_z} * M_y \\wy &= -D * \text{gamma} * M_{t_z} * M_x \\wz &= 0\end{aligned}$$

To calculate this interaction, the BlochParams must be of the density type, as the ‘real’ magnetization of the sample cell must be calculated. Setting the Magnetization to 1 for the spin (the Particle case) would cause an abnormally large effect for this interaction.

Currently there is no off axis ($B_0 \neq B_z$) interaction as this is designed as an approximation to a liquid NMR experiment.

BulkSus::constructor

Function:

- BulkSus()**

Input:

- void

Description:

- The empty constructor, sets D=0

Example Usage:

- BulkSus myBs;

BulkSus::constructor

Function: `□ BulkSus(double D)`

Input: `□ D--> a number from 0..1 representing the reaction of the sample to the applied B field`

Description: `□ The basic constructor, sets the D parameter to the inputted value.`

Example Usage: `□ BulkSus myBs(0.04);`

BulkSus::element extraction

Function:

□ **double D()**

Input:

□ **void**

Description:

□ returns the current value of the susceptibility parameter

Example Usage:

□ **BulkSus myBs(0.5);
cout<<myBs.D();**

BulkSus::assignments

Function: **void D(double newD);**

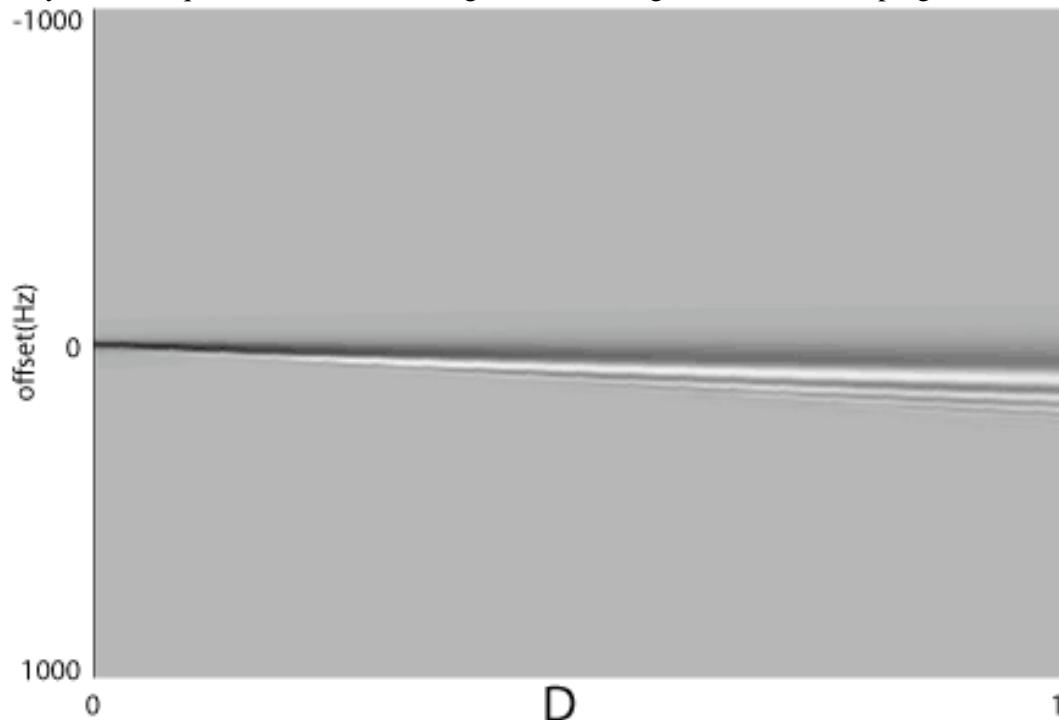
Input: newD--> a number from 0..1 representing the reaction of the sample to the applied B field

Description: Sets the susceptibility parameter.

Example Usage: BulkSus myBs;
 myBs.D(0.05);

The effect of the Bulk susceptibility

This examples simply loops through a series of Bulk susceptibility parameters over a high magnetization (5 moles, $B_0=11$ Telsa) spin system to see the effect on the offset. As D increases, the offsets should 'chirp' more and more as they relax to equilibrium. The below figure is the data generated from the program, and displays the expected result.



```
#-----INput File-----
#parameter file for looping through
# several BulkSus parameters

dim 1,1,1
min -0.5,-0.5,-0.5
max 0.5, 0.5, 0.5

#fid pieces
npts 1024
tf 2

#the pulse bits
pulseangle 90
pulsephase 270
pulseamp 80000

#basic spin parameters
Bo 11
temperature 300
offset 0

T2 0.1
T1 0.1

spintype 1H
moles 5
```

```

#the number of D's to loop through
#for the Bulk Suseptibility
D 0
Dstep 0.05
numD 20

#file output names for the data
fidout data

-----end INput File-----

#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

timer stopwatch;
void printTime(int nrounds=1){
    std::cout <<std::endl<< "Time taken: "
    << (stopwatch()/nrounds) << " seconds";
}

void Info(std::string mess)
{
    cout<<mess<<endl;
    cout.flush();
}

int main(int argc,char* argv[])
{
    std::string fn;

//the parameter file
    query_parameter(argc,argv,1, "Enter file to parse: ", fn);
    Parameters pset(fn);

//get the basic parameters
    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");
    double inTemp=pset.getParamD("temperature");
    string spintype=pset.getParamS("spintype");
    string detsp=spintype;
    double t2s=pset.getParamD("T2");
    double t1s=pset.getParamD("T1");

    double moles=pset.getParamD("moles");

    std::string fout=pset.getParamS("fidout");

    coord<int> dims(pset.getParamCoordI("dim"));
    coord<> mins(pset.getParamCoordD("min"));
    coord<> maxs(pset.getParamCoordD("max"));

    std::string dataou=pset.getParamS("trajectories", "", false);
}

```

```

// Grid Set up
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrid;

Info("Creating grid....");
Grid<UniformGrid> gg(mins, maxs, dims);

Info("Creating initial shape....");
TheShape tester;
Info("Creating total shape-grid....");
TheGrid jj( gg, tester);

//List BlochParameters
typedef ListBlochParams<
    TheGrid,
    BOptions::Density | BOptions::HighField,
    double > MyPars;
int nsp=jj.size();
Info("Creating entire spin parameter list for "+itost(nsp)+" spins....");
MyPars mypars(nsp, "1H", jj);
nsp=mypars.size();

//The pulse list for a real pulse on protons..
Info("Creating real pulse lists...");

//get the info from the pset
double pang=pset.getParamD("pulseangle");
double amp=pset.getParamD("pulseamp");
double phase=pset.getParamD("pulsephase");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype, amp*PI2, phase*DEG2RAD);

//get the Bo
double inBo=pset.getParamD("Bo");

Info("Setting spin parameter offsets....");
for(int j=0;j<nsp;j++){
    mypars(j)=spintype;
    mypars(j).moles(moles);
    //set the proper Bo
    mypars(j).Bo(inBo);
    mypars.temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(cout);
PP1.print(cout);

//get the time for the 90 pulse
double tpulse=PP1.timeForAngle(pang*Pi/180., spintype);

//the time trains
Info("Initializing Time train for first Pulse....");
TimeTrain<UniformTimeEngine >

```

```

P1(UniformTimeEngine(0., tpulse, 10,100));

Info("Initializing Time train for FID....");
TimeTrain<UniformTimeEngine >
F1(UniformTimeEngine(tpulse, tpulse+tf, nsteps,20));

//Extra interactions
typedef Interactions<
    Offset<>,
    Relax<>,
    BulkSus > MyInteractions;
Info("Setting Interactions....");

//the offsets
//get the first offset
double offset=pset.getParamD("offset")*PI2;
Offset<> myOffs(mypars, offset);

//Relaxation
Relax<> myRels(mypars, (!t2s)?0.0:1.0/t2s, (!t1s)?0.0:1.0/t1s);

//Bulk suseptibility
double D=pset.getParamD("D");
double Dstp=pset.getParamD("Dstep");
int nDs=pset.getParamI("numD");

BulkSus myBs(D);

//total interaction obect
MyInteractions MyInts(myOffs, myRels, myBs);

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//our data matrix
matrix FIDs(nDs, nsteps);

//loop over all our D values
for(int kk=0;kk<nDs;++kk){

    //THis is the BLoch solve to perform a pulse
    Info("Initializing total parameter list with a pulse....");
    PulseBloch myparspulse(mypars, PP1, MyInts);

    //This is the Bloch solver to Collect the FID (i.e. has no pulses...FASTER)
    Info("Initializing total parameter list for FID collection....");
    NoPulseBloch me;
    me=(myparspulse);
    Info("Integrating first Pulse....");

    //out initial condition
    Vector<coord<> > tm=me.currentMag();

    stopwatch.reset();
    BlochSolver<PulseBloch > drivP(myparspulse, tm, "out");

    //integrate the Pulse
}

```

```

    drivP.setWritePolicy(Hold);
    if(!drivP.solve(P1)){
        Info(" ERROR!!..could not integrate pulse P1....");
        return -1;
    }

    //the fids initial condition is just the previous
    // integrations last point
    BlochSolver<NoPulseBloch> driv(me, drivP.lastPoint());
    Info("Integrating FID ....");

    //set various data collection policies
    std::string lypname="lyps";
    if(dataou!=""){
        driv.setCollectionPolicy(All);
    }else{
        drivP.setCollectionPolicy(MagAndFID);
    }
    driv.setWritePolicy(Hold);

    //set the detection spin
    driv.setDetect(detsp);

    //integrate the FID
    if(driv.solve(F1)){
        FIDs.putRow(kk, driv.FID());
    }
    //update the BulkSus interaction
    myBs.D(myBs.D()+Dstp);

}

matstream matout(fout, ios::binary | ios::out);
matout.put("vdat", FIDs);
Vector<double> dlist(Spread<double>(D, (nDs*Dstp)+D, Dstp));
matout.put("ds", dlist);
matout.close();
printTime();
}

```

--Constructors

--- RadDamp

--- RadDamp

--Element Extraction

--- tr

--Assignments

--- tr

Examples

--- simple RadDamp

class RadDamp { ...

Radiation damping is another high field effect that relies on the bulk magnetization of the sample. The moving magnetization of the sample induces a field fluctuation in the detection coil. The detection coil then induces a field back on the sample, etc., etc. There are many complex theoretical descriptions for both the quantum mechanical and classical realms. But classically the interaction can be decomposed into a simple magnetization conserving nonlinear 'relaxation.' It is magnetization conserving because the total magnetization is a constant of motion for this interaction, it is a relaxation because it tends to push the magnetization make towards equilibrium (the B_0 axis). Its strength depends on the total magnetization and the coil sensitivity (the coil's Q factor). These can be simply translated into a rate, t_r . If Mt is the total magnetization, this interaction adds these terms to the rates of our master equation of motion.

$$\begin{aligned}w_x &= -1/(|Mt|) * Mt_x * M_z \\w_y &= -1/(|Mt|) * Mt_y * M_z \\w_z &= -1/(|Mt|) * (Mt_x * M_x + Mt_y * M_y)\end{aligned}$$

Because this interaction is typically only present in highly magnetized, high sensitive detection coils, off axis interactions are not included in this interaction (as the conditions described are a high field liquid experiments). And because it requires the sample magnetization, this typically makes sense for `BOptions::Density`. However, it is normalized with respect to the total magnetization, $|Mt|$, `BOptions::Particle` will also behave properly.

RadDamp::constructor

Function:

□ **RadDamp()**

Input:

□ void

Description:

□ Empty constructor, sets tr=0

Example Usage:

□ `RadDamp myrd;`

RadDamp::constructor

Function: **RadDamp(double tr)**

Input: tr--> the radation damping time constant (in seconds)

Description: Constructor that sets the Radiation damping time constant (in seconds)

Example Usage: RadDamp myRD(0.07);

RadDamp::element extraction

Function:	<input type="checkbox"/> double tr()
Input:	<input type="checkbox"/> void
Description:	<input type="checkbox"/> Returns the radation damping coupling constant in seconds.
Example Usage:	<input type="checkbox"/> <code>RadDamp myRd(0.07);</code> <code>cout<<myRd.tr(); //prints 0.07</code>

RadDamp::assignments

Function: `void tr(double newTr)`

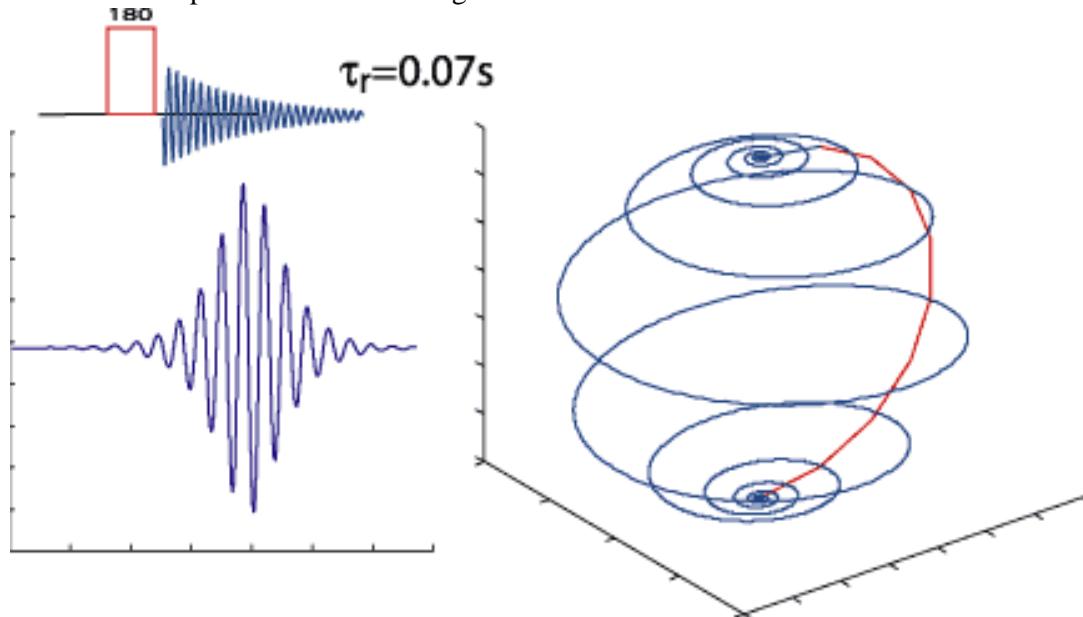
Input: `newTr-->` the new value for the Radiation damping time constant in seconds

Description: `Sets the Radiation damping time constant in seconds.`

Example Usage: `RadDamp myRd(0.07);
myRd.tr(0.08);`

The classic radiation damping experiment.

This example simulates the classic radiation damping experiment. If no relaxation is present, a 180 degree pulse to the spins will simply invert the spins and the system will stay there forever producing only residual signal (because of pulse imperfections). With radiation damping included the system has a tendency to be driven back to the equilibrium z-axis. The effect produces what looks to be an echo. The trajectory shown below also shows you that the interaction preserves the total magnetization.



```
#-----INput File-----
#parameter file for a simple Radation Damping example

dim 1,1,10
min -0.5,-0.5,-0.5
max 0.5, 0.5, 0.5

#fid pieces
npts 1024
tf 2

#the pulse bits
pulseangle 180
pulsephase 270
pulseamp 80000

#basic spin parameters
Bo 11
temperature 300
offset 30

T2 0.0
T1 0.0

spintype 1H
moles 1

#the Radation Damping strength in seconds
tr 0.07
```

```

#file output names for the data
fidout data
magout mag
trajectories traj

-----end INput File-----

#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

/*
A simple 90 pulse demonstrating the
effect of radiation dmaping on a few spins
*/

timer stopwatch;
void printTime(int nrounds=1){
    std::cout << std::endl << "Time taken: "
    << (stopwatch() / nrounds) << " seconds";
}

void Info(std::string mess)
{
    cout << mess << endl;
    cout.flush();
}

int main(int argc, char* argv[])
{
    std::string fn;

//the parameter file
    query_parameter(argc, argv, 1, "Enter file to parse: ", fn);
    Parameters pset(fn);

//get the basic parameters
    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");
    double inTemp=pset.getParamD("temperature");
    string spintype=pset.getParamS("spintype");
    string detsp=spintype;
    double t2s=pset.getParamD("T2");
    double t1s=pset.getParamD("T1");

    double moles=pset.getParamD("moles");

    std::string fout=pset.getParamS("fidout");
    std::string magout=pset.getParamS("magout");

```

```

coord<int> dims(pset.getParamCoordI("dim"));
coord<> mins(pset.getParamCoordD("min"));
coord<> maxs(pset.getParamCoordD("max"));

std::string dataou=pset.getParamS("trajectories", "", false);

// Grid Set up
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrid;

Info("Creating grid....");
Grid<UniformGrid> gg(mins, maxs, dims);

Info("Creating initial shape....");
TheShape tester;
Info("Creating total shape-grid....");
TheGrid jj( gg, tester);

//List BlochParameters
typedef ListBlochParams<
    TheGrid,
    BPOptions::Particle | BPOptions::HighField,
    double > MyPars;
int nsp=jj.size();
Info("Creating entire spin parameter list for "+itost(nsp)+" spins....");
MyPars mypars(nsp, "1H", jj);
nsp=mypars.size();

//The pulse list for a real pulse on protons..
Info("Creating real pulse lists...");

//get the info from the pset
double pang=pset.getParamD("pulseangle");
double amp=pset.getParamD("pulseamp");
double phase=pset.getParamD("pulsephase");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype, amp*PI2, phase*DEG2RAD);

//get the Bo
double inBo=pset.getParamD("Bo");

Info("Setting spin parameter offsets....");
for(int j=0;j<nsp;j++){
    mypars(j)=spintype;
    mypars(j).moles(moles);
    mypars(j).Bo(inBo);
    mypars.temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(cout);
PP1.print(cout);

//get the time for the 90 pulse

```

```

double tpulse=PP1.timeForAngle(pang*Pi/180., spintype);

//the time trains
Info("Initializing Time train for first Pulse....");
TimeTrain<UniformTimeEngine >
    P1(UniformTimeEngine(0., tpulse, 10,100));

Info("Initializing Time train for FID....");
TimeTrain<UniformTimeEngine >
    F1(UniformTimeEngine(tpulse, tpulse+tf, nsteps,20));

//Extra interactions
typedef Interactions<
    Offset<>,
    Relax<>,
    RadDamp > MyInteractions;
Info("Setting Interactions....");

//the offsets
//get the first offset
double offset=pset.getParamD("offset")*PI2;
Offset<> myOffs(mypars, offset);

//Relaxation
Relax<> myRels(mypars, (!t2s)?0.0:1.0/t2s, (!tls)?0.0:1.0/tls);

//Bulk suseptibility
double tr=pset.getParamD("tr");

RadDamp myRD(tr);

//total interaction obect
MyInteractions MyInts(myOffs, myRels, myRD);

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//THis is the BLoch solve to perform a pulse
Info("Initializing total parameter list with a pulse....");
PulseBloch myparspulse(mypars, PP1, MyInts);

//This is the Bloch solver to Collect the FID (i.e. has no pulsles...FASTER)
Info("Initializing total parameter list for FID collection....");
NoPulseBloch me;
me=(myparspulse);
Info("Integrating first Pulse....");

//our initial condition
Vector<coord<> > tm=me.currentMag();

stopwatch.reset();
BlochSolver<PulseBloch > drivP(myparspulse, tm, "out");

```

```

//output trajectory data if wanted
    std::ofstream trajout;
    if(dataou!=""){
        trajout.open(dataou.c_str());
        drivP.setCollectionPolicy(All);
    }else{
        drivP.setCollectionPolicy(FinalPoint);
    }

//integrate the Pulse
    drivP.setWritePolicy(Hold);
    if(!drivP.solve(P1)){
        Info(" ERROR!!..could not integrate pulse P1....");
    }
    return -1;
}else if(dataou!=""){
    drivP.writeData(trajout);
}

//the fids initial condition is just the previous
// integrations last point
    BlochSolver<NoPulseBloch> driv(me, drivP.lastPoint());

    Info(
Integrating FID ....
);

//set various data collection policies
    std::string lypname="lyps";
    if(dataou!=""){
        driv.setCollectionPolicy(All);
    }else{
        drivP.setCollectionPolicy(MagAndFID);
    }
    driv.setWritePolicy(Hold);

//set the detection spin
    driv.setDetect(detsp);

//integrate the FID
    if(driv.solve(F1)){
        //dump out the data
        driv.writeSpectrum(fout);
        driv.writeMag(magout);
        if(dataou!="") driv.writeData(trajout);
    }

printTime();

}

```


--Constructors**--- DipoleDipole****--- DipoleDipole****--Element Extraction****--- factor****--- grid****--Assignments****--- setFactor****--- setGrid****--Other Functions****--- off, on****Examples****--- jeener cube**

```
template<class Grid_T>
class DipoleDipole {..
```

```
template<class Grid_T>
class DimLessDipole {..
```

The dipole-dipole is one of the most important interactions in NMR. It allows for long range coherence transfers to get distance information between atoms where J-couplings are negligible. Recent developments have shown that the residual dipole couplings can lead to interesting behavior and the possible extraction of even more system information. Classically, the dipole-dipole interaction remains a highly non-linear interaction (in quantum mechanics this nonlinearity seems to disappear at a first approximation) especially for non-symmetric sample shapes.

The dipolar fields from each spin contribute to the total effect seen on one spin, making this interaction one of the most expensive interactions to calculate going as N^2 where N is the number of spins. The other interactions simply scale as N . The magnetic field at a point r from a dipole at position r_i is given as

$$B(r) = \mu_0((M) - (M \cdot \hat{r}) * \hat{r}) / (4\pi |r-r_i|^3)$$

where μ_0 is the permittivity of a vacuum, and \hat{r} is the unit vector in the r direction. In a high magnetic field, the only direction we need to be interested in is the direction along the main magnetic field. For a $B_0 = (0, 0, B_z)$ (or $\hat{r} = (0, 0, 1)$ in Cartesian space or $\hat{r} = (0, 0, \cos(\theta))$ in spherical space), the above equation reduces to

$$\begin{aligned} B_x(r) &= -\mu_0 / (4\pi |r-r_i|^3) M_x \\ B_y(r) &= -\mu_0 / (4\pi |r-r_i|^3) M_y \\ B_z(r) &= \mu_0 (1 + 3 \cos(2\theta)) / (8\pi |r-r_i|^3) M_z \end{aligned}$$

This looks much like the high field dipolar Hamiltonian used often in quantum mechanics. Here θ is the angle formed between the z -axis and the r vector ($\cos(\theta) = z / |r-r_i|$). We use this equation when our BlochParameters offset type is simply a double (the default high field along the z -axis case). When the Magnetic field is no longer along the z -axis, but some off axis

$$B_0 = (bx, by, bz),$$

we must truncate the interaction with respect to that new axis, thus our rhat vector then becomes

```
rhat=1/|Bo|(bx, by, bz) (in cartesian
space)
or in spherical cords
rhat_x 1/|Bo| B_x (B_z cos(theta) + B_x
cos(phi) sin(theta) + B_y sin(phi)
sin(theta))
rhat_y= 1/|Bo| B_y (B_z cos(theta) + B_x
cos(phi) sin(theta) + B_y sin(phi)
sin(theta))
rhat_z= 1/|Bo| B_z (B_z cos(theta) + B_x
cos(phi) sin(theta) + B_y sin(phi)
sin(theta))
```

which makes the equations slightly more complicated

$$B_x(r) = \mu_0 / (4\pi(3 B_x ((B_x M_x + B_y M_y + B_z M_z) ((B_z \cos(\theta) + ((B_x \cos(\phi) + B_y \sin(\phi))) \sin(\theta)))^2 - M_x)) / |r - r_i|^3$$

$$B_y(r) = \mu_0 / (4\pi(3 B_y ((B_x M_x + B_y M_y + B_z M_z) ((B_z \cos(\theta) + ((B_x \cos(\phi) + B_y \sin(\phi))) \sin(\theta)))^2 - M_y)) / |r - r_i|^3$$

$$B_z(r) = \mu_0 / (4\pi(3 B_z ((B_x M_x + B_y M_y + B_z M_z) ((B_z \cos(\theta) + ((B_x \cos(\phi) + B_y \sin(\phi))) \sin(\theta)))^2 - M_z)) / |r - r_i|^3$$

where $\phi = \text{atan}(x/y)$ where $x = (r_x - r_{xi})$ and $y = (r_y - r_{yi})$

From these equations, we have the generic dipole coupling between spins in a high magnetic field. The above form is useful for plotting what the magnetic field looks like in real space, however, simple application of the original formula (the first one above) give you the same result, and with minimal computation.

The above equation rely explicitly on the distance between the spin, and hence need a grid to determine this distance. Sometimes there is the need to specify the maximal dipolar coupling between spins. This can be achieved using the `DimLessDipole` class, where you can specify a single maximal coupling for the closest (nearest neighbor) spins in Hz (this effectively replaces the $\mu_0/4\pi r^3$ term in the above equations). The remaining coupling will be scaled according to the cube of the distance away. Currently there is no built in mechanism to specify only nearest neighbor interactions. This may change in another release using the 'StencilPrep' object. If you have a

time dependant field off axis field, then you must include an Offset class in the interaction list BEFORE the DipoleDipole class. The Offset class will set the proper Bo field for each spin, which the DipoleDipole class will then use to determine the proper angles.

DipoleDipole::constructor

Function: `DipoleDipole<Grid_T>()`
 `DimLessDipole<Grid_T>()`

Input: void

Description: The empty constructor, sets the grid pointer to NULL and the factor in DimLessDipole to 2π rad/second

Example `DipoleDipole myDip;`
Usage:

DipoleDipole::constructor

- Function:**
- `DipoleDipole<Grid_T>(Grid_t &gr)`
 - `DimLessDipole<Grid_T>(Grid_t &gr, double fact=PI2);`
- Input:**
- gr--> a valid grid, or XZshape object
 - fact--> the maximal coupling for any given set of spins in rad/sec (valid for DimLessDipole only)
- Description:**
- The basic constructor that sets the grid pointer to the input grid. The DimLessDipole has an extra 'factor' parameter that determines the maximal coupling between any set of 2 spins, the default is 2*pi rad/sec.
- Example**
- `coord<> mins(-1,-1,-1), maxs(1,1,1);`
- Usage:**
- `coord<int> dims(10,10,10);`
- `Grid<UniformGrid> mygrid(mins, maxs, dims);`
- `XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a rectangular grid`
- `DimLessDipole<XYZshape<XYZrect> > myDip(myshape, 1000.0);`

DipoleDipole::element extraction

Function: `double factor()`

Input: `void`

Description: `Returns the maximal coupling in Rad/Sec for a DimLessDipole object.`

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
```

```
DimLessDipole<XYZshape<XYZrect> > myDip(myshape, 1000.0*PI2);
```

```
cout<<myDip.factor(); //prints 1000.0*PI2
```

DipoleDipole::element extraction

Function: `Grid_T *grid();`

Input: `void`

Description: `Returns the pointer to the current grid being used.`

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
```

```
DimLessDipole<XYZshape<XYZrect> > myDip(myshape, 1000.0);
XYZshape<XYZrect> *aGrid=myDip.grid();
```

DipoleDipole::assignments

Function: `void setFactor(double newfact);`

Input: `newfact`--> the new maximal dipolar coupling in rad/second

Description: `Sets the maximal dipolar coupling (in rad/seconds) between the closest two spins for the DimLessDipole object. The rest of the dipolar couplings will be scaled based on the cube of their distance away.`

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
```

```
DimLessDipole<XYZshape<XYZrect> > myDip(myshape, 1000.0*PI2);
myDip.setFactor(2000.0*PI2);
```

DipoleDipole::assignments

Function: `void setGrid(Grid_T &ing);`

Input: `ing`--> a new grid object of type `Grid_T`

Description: `ing` Sets the grid for the dipole interaction. It sets a pointer to the grid object, so if the grid is destroyed before the Dipole object, it will perhaps crash the program.

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
```

```
DimLessDipole<XYZshape<XYZrect> > myDip
myDip.setGrid(myshape);
```

DipoleDipole::other

Function: **void off();**
 void on();

Input: void

Description: This turns the coupling on or off. If it is off then the interaction will not be calculated, if it on it will calculate the dipolar interaction.

Example coord<> mins(-1,-1,-1), maxs(1,1,1);

Usage: coord<int> dims(10,10,10);

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid

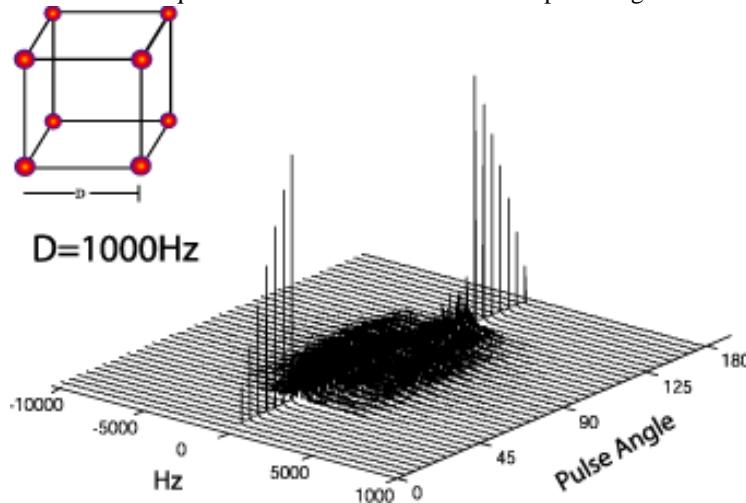
DimLessDipole<XYZshape<XYZrect> > myDip(myshape, 1000.0);

//turn it off
myDip.off();

//turn it back on
myDip.on();
```

This is a replication of Jeener's simulation paper in PRL ("Dynamical effects of the dipolar field inhomogeneities in high-resolution NMR: Spectral clustering and instabilities", Jeener J, PHYSICAL REVIEW LETTERS, 82 (8): 1772-1775 FEB 22 1999).

Here he explores the dipolar instabilities as a function of pulse angle of a cube (8) of spins. For angles between 0..45 we see 'spectral clustering' where even though we have many dipolar frequencies present, the dynamics seem to indicate a single evolution frequency, however for angles 45..125 we see instabilities, from 125..180 we see the spectral clustering again. The simulation below shows this effect in the frequencies of the simulation for each pulse angle.



```
#----- The Input File -----
#parameter file for 2 pulse sequences

dim 2,2,2
min -0.5,-0.5,-0.5
max 0.5, 0.5, 0.5

#fid pieces
npts 4096
tf 1

#the pulse bits
pulseangle 0
pulsephase 270
pulseamp 100000
pulsesteps 41
pulsestepsize 4.5

#basic spin parameters
Bo 11
temperature 300
offset 0
T2 0
T1 0
spintype 1H
moles .2

#the extra interactions parts
dipole_str 50

#calculate lypuvnovs[1] or not[0]
lyps 0
lypout lyps

#file output names for the data
fidout data
```

```

#-----end InputFile -----
#include "blochlib.h"

//the required 2 namespaces
using namespace BlochLib;
using namespace std;

/*
loops through various pulse angles
on a dipole-dipole coupled system using the DimLessDipole object
offsets, and relaxation parameters
*/

timer stopwatch;
void printTime(int nrounds=1){
    std::cout << std::endl << "Time taken: " << (stopwatch() / nrounds) << " seconds";
}

void Info(std::string mess)
{
    cout << mess << endl;
    cout.flush();
}

int main(int argc, char* argv[])
{
    std::string fn;
    query_parameter(argc, argv, 1, "Enter file to parse: ", fn);
    Parameters pset(fn);

    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");
    double offset=pset.getParamD("offset")*PI2;
    double inBo=pset.getParamD("Bo");
    double inTemp=pset.getParamD("temperature");
    string spintype=pset.getParamS("spintype");
    string detsp=spintype;
    double t2s=pset.getParamD("T2");
    double t1s=pset.getParamD("T1");
    double moles=pset.getParamD("moles");
    double dipstr=pset.getParamD("dipole_str")*PI2;

    std::string fout=pset.getParamS("fidout");

    coord<int> dims(pset.getParamCoordI("dim"));
    coord<> mins(pset.getParamCoordD("min"));
    coord<> maxs(pset.getParamCoordD("max"));

    int cv=pset.getParamI("lyps");

    // Bloch set up testing

    typedef XYZfull TheShape;
    typedef XYZshape<TheShape> TheGrid;

    Info("Creating grid....");
    Grid<UniformGrid> gg(mins, maxs, dims);

    Info("Creating initial shape....");
    TheShape tester;
}

```

```

Info("Creating total shape-grid....");
TheGrid jj( gg, tester);

typedef ListBlochParams< TheGrid,
    BPOptions::Particle | BPOptions::HighField,
    double > MyPars;
int nsp=jj.size();
Info("Creating entire spin parameter list for "+itost(nsp)+" spins....");
MyPars mypars(nsp, "1H", jj);
nsp=mypars.size();

Info("Setting spin parameter offsets....");
for(int j=0;j<nsp;j++){
    mypars(j)=spintype;
    mypars(j).moles(moles);
    mypars(j).Bo(inBo);
    mypars.temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(cout);

//time train testing

//Extra interactions
typedef TanhScale Scaler;
typedef Interactions<Offset>,
    Relax<>,
    DimLessDipole<TheGrid> > MyInteractions;
Info("Setting Interactions....");
Offset<> myOffs(mypars, offset);
Relax<> myRels(mypars, (!t2s)?0.0:1.0/t2s, (!t1s)?0.0:1.0/t1s);
DimLessDipole<TheGrid> DipDip(jj, dipstr);

MyInteractions MyInts(myOffs, myRels, DipDip);

//the pulse object
//The pulse list for a real pulse on protons..
double pang=pset.getParamD("pulseangle");
double pstep=pset.getParamD("pulsestepsize");
int numP=pset.getParamI("pulsesteps");
double amp=pset.getParamD("pulseamp");
double phase=pset.getParamD("pulsephase");

Info("Creating real pulse lists...");

// (spin, amplitude, phase, offset)
Pulse PP1(spintype, amp*PI2, phase*DEG2RAD);
PP1.print(cout);

//data FID
matrix FIDs(numP, nsteps);

for(int kk=0;kk<numP;++kk)
{
    double tpulse=PP1.timeForAngle((pang+double(kk)*pstep)*Pi/180., spintype);
    std::cout<<std::endl<<"On Pulse Angle: "
        <<(pang+double(kk)*pstep)<<" degrees "<<std::endl;
    Info("Initializing Time train for first Pulse....");
    TimeTrain<UniformTimeEngine > P1(UniformTimeEngine(0., tpulse, 10,100));
}

```

```

Info("Initializing Time train for FID....");
TimeTrain<UniformTimeEngine > F1(UniformTimeEngine(tpulse, tpulse+tf, nsteps,20));

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//THis is the BLoch solve to perform a pulse
Info("Initializing total parameter list with a pulse....");
PulseBloch myparspulse(mypars, PP1, MyInts);
if(cv) myparspulse.calcVariational();

if(dipstr==0) DipDip.Off();

//This is the Bloch solver to Collect the FID (i.e. has no pusles...FASTER)
Info("Initializing total parameter list for FID collection....");
NoPulseBloch me;
me=(myparspulse);
Info("Integrating first Pulse....");
Vector<coord> tm=me.currentMag();

stopwatch.reset();
BlochSolver<PulseBloch > drivP(myparspulse, tm);

drivP.setCollectionPolicy(FinalPoint);

drivP.setWritePolicy(Hold);
if(!drivP.solve(P1)){
    Info(" ERROR!!!..could not integrate pulse P1....");
    return -1;
}

BlochSolver<NoPulseBloch > driv(me, drivP.lastPoint());

Info("Integrating FID ....");

std::string lypname="lyps";
drivP.setCollectionPolicy(MagAndFID);

driv.setWritePolicy(Hold);
driv.setDetect(detsp);
if(cv) {
    driv.setLyapunovPolicy(LypContinous);
    driv.setLypDataFile(lypname);
}

if(driv.solve(F1)){
    FIDs.putRow(kk, driv.FID());
}
}

matstream matout(fout);
matout.put("vdat", FIDs);
Vector<double> pangs(Spread<double>(pang, pang+(numP*pstep), pstep));
matout.put("pangs", pangs);
matout.close();
printTime();
}

```

--Constructors	template<class Grid_T>
--- DemagField	class DemagField{...
--- DemagField	
--- ModulatedDemagField	class ModulatedDemagField{...
--Element Extraction	
--- direction	
--- td	
--Assignments	
--- setDirection()	
--- setTd	
Examples	
--- YY Lin	

The exploration of the demagnetizing field in NMR is a relatively recent development (E. Belorizky, et al. chem phys. lett. Volume 175, Issue 6, 28 December 1990, Pages 579-584). Its arrival into the NMR scene so late is a consequence of the higher and higher field strengths available to the NMR spectroscopist. In real samples this effect is quite small, but still present enough to create potentially create long range coherences (He QH, et al. J chem phys 98 (9): 6779-6800 may 1 1993). It has seen recent interest (as of this writing see Huang SY, et al. J Chem.. Phys. 116 (23): 10325-10337 JUN 15 2002) in potentially chaotic dynamics. Ideas are currently brewing about dynamic control and long range (macroscopic range) distance determination using this highly non-linear effect.

This effect is essentially a better theoretical treatment of the dipole-dipole interaction of a macroscopic sample. Instead of a direct spin-spin interaction, we sum over all the spins inside a volume to determine the dipole field at a point, r . The magnetic field induced by a cell volume can be written

$$dB = \mu_0 ((M(r)) - (M(r) \cdot r\hat{r}) * r\hat{r}) / (4\pi |r-r_i|^3) dr^3$$

To get the entire field at point r we must perform a very hard integral. For highly symmetric samples (like a sphere) this interaction is 0, for almost any other geometry, this interaction is non-zero. For uniformly magnetized samples this interaction is also 0, but application of gradients, or even other temperature gradients, will cause this interaction to be non-zero. In our Bloch Interaction frame work, we can approximate the integral by summing over the cell volumes in the grid. For small grids, this sum is a poor approximation to the interaction, however, it is the best we have for the general case.

This interaction can be treated in much the same way the dipole-dipole interaction is except for addition of the dr^3 .

There are a few approximations that we can make given the physical description of our system. The first is the assumption that the magnetization of the sample is uniform throughout the entire sample. When this happens we can factor out the $M(r)$ of the above integral, and we are then left with a shape factor.

$$B = \mu_0 * D / 3 * (M - (M \cdot r\hat{r}) * r\hat{r})$$

This shape factor, D , is very hard to calculate, so it is left as an input parameter. For a cylinder $D \sim 1/2$, for a sphere $D=0$, for a flat disk $D=1$.

This is essentially the Bulk Susceptibility. However, in most any normal situation, the assumption that M is uniform everywhere is quite false and the full integral equation must be used.

There is one more useful approximation involves to opposite extreme, when the total magnetization is zero. One can create such a situation by creating a fully modulated magnetization along a single direction (a full helix using gradient pulses). Then, via a few Fourier transform enlightenments (Deville et al, Phys Rev B (19) p. 5666 1979) we have a much simpler form

$$B = \text{del} / (\text{td} * 3) * (M - (M . rhat) * rhat)$$

where 'del' is $(3(s.rhat)^2 - 1)/2$. 's' is the direction of the magnetization modulation. So for a Z-gradient helix, we have $s=zhat$, and usually our high field makes $rhat=zhat$. 'td' is the basic magnitude of the effect and is related to the total magnetization present ($\text{td}=1/(\gamma \mu_0 M_0)$). Using this equation as is yields a subtle error. The error comes from the Fourier Transform analysis, where any global constant field for a singularity, thus to fix this we simply subtract out the average magnetization at any given time

$$B = \text{del} / (\text{td} * 3) * ((M - \langle M \rangle) - ((M . rhat) - (\langle M \rangle . rhat)) * rhat)$$

This interaction is referred to as the 'ModulatedDemagField' class. And it does not need a grid, simply the total magnetization, making this interaction light years faster than the 'normal' case. It is also a normalized interaction meaning both BPOptions::Particle or BPOptions::Density will work. However, when you calculate td for the BPOptions::Particle, you may be tempted to use the 'totalMo()' function in ListBlochParams. This will return the wrong desired value (instead of Mo~0.06 for a 600 MHz magnet and 104 mmol protons at 300 K, you will get the sum of the total number of spins in your simulation), so you will need to calculate the Mo properly.

DemagField::constructor

Function: **DemagField<Grid_T>()**
 ModulatedDemagField()

Input: void

Description: The empty constructor. For the DemagField class, the grid point is NULL, for the ModulatedDemagField, $td=0$ and $s=(0,0,1)$.

Example `DemagField<myGrid_T> myDM; //empty demagfield`

Usage: `ModulatedDemagField myMDM; //empty modulated field`

DemagFied::constructor

Function: `DemagField<Grid_T>(Grid_T &ingrid)`

Input: `ingrid-->` a grid object of some kind

Description: `Sets up a basic object using the input grid to calculate both the |r|^3 and the cellvolumes. It sets a grid pointer inside the object to ingrid, thus if you destroy the grid before this object, you will perhaps crash the program.`

Example `coord<> mins(-1,-1,-1), maxs(1,1,1);`

Usage: `coord<int> dims(10,10,10);`

```
Grid<UniformGrid> mygrid(mins, maxs, dims);
XYZshape<XYZrect> myshape(mygrid, XYZrect(mins, maxs, dims)); //a
rectangular grid
```

```
DemagField<XYZshape<XYZrect> > myDF(myshape);
```

DemagFied::constructor

Function: `ModulatedDemagField(double td, coord<> s=(0,0,1));`

Input: `td-->` the time constant (in seconds) typically $1/(\mu_0 \gamma M_0)$
`s-->` the modulated magnetization direction. (this input vector will be normalized on input)

Description: `□` The basic constructor for the ModulatedDeamgField object. It sets the time constant to `td` and the modulated direction to `s` (the default is the z-axis).

Example `ModulatedDemagField myMDF(0.04); //uses the default direction`

Usage: `ModulatedDemagField myMDF(0.04, coord<>(0,1,1)); //a different direction`

DemagFied::element extraction

Function: `coord<> direction()`

Input: `void`

Description: `Returns the current direction of the modulated magnetiation for the ModulatedDemagField class.`

Example `ModulatedDemagField myMDF(0.04, coord<>(0,1,1)); //a different direction`

Usage: `cout<<myMDF.direction()<<endl;`

DemagFied::element extraction

Function: **double td()**

Input: void

Description: Returns the current time constant (in seconds) for the ModulatedDemagField class.

Example Usage: `ModulatedDemagField myMDF(0.04); //uses the default direction
cout<<myMDF.td()<<endl;`

DemagFied::assignments

Function: `void setDirection(coord<> s);`

Input: `s-->` the new Modulated direction.

Description: `Sets the new direction in the ModulatedDemagField. The program will normalize this vector upon input.`

Example

```
-----  
----ModulatedDemagField myMDF(0.04); //uses the default direction  
myMDF.setDirection(coord<>(2,3,3)); //sets the new direction
```

DemagFied::assignments

Function: `void setTd(double td)`

Input: `td-->` the new time constant (in seconds)

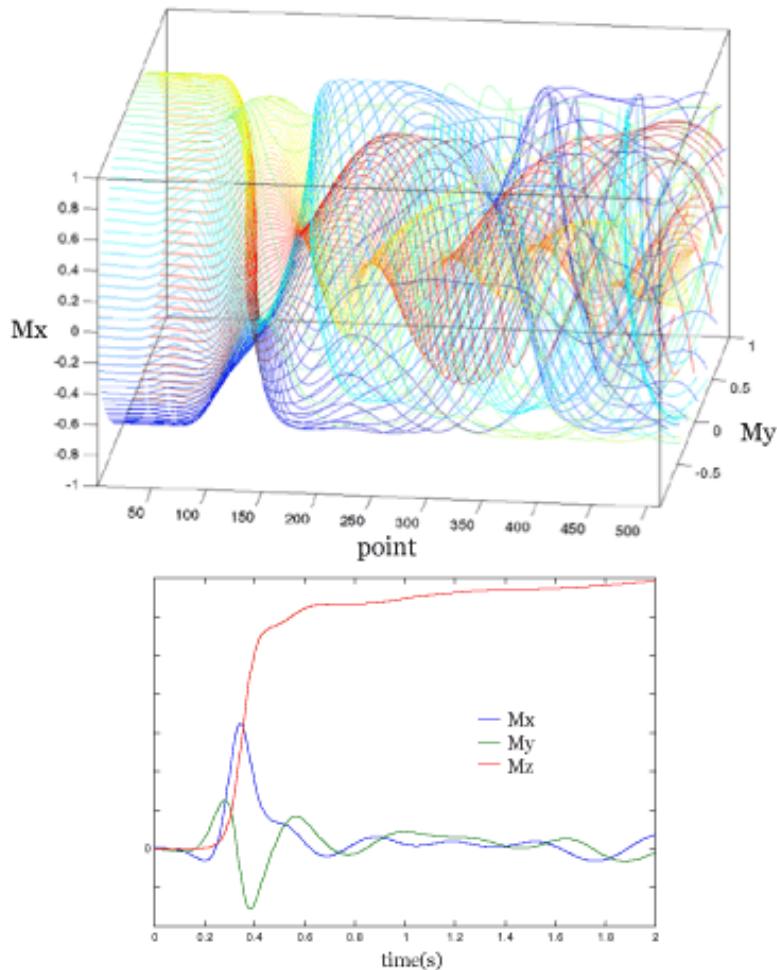
Description: `Sets the time constant in ModulateDemagField. It should be >=0.`

Example `ModulatedDemagField myMDF(0.04, coord<>(0,1,1)); //a different direction`

Usage: `myMDF.setTd(0.06); //reset the time constant`

This simulation represents one of the more interesting features of the DemagField and its interaction with Radiation Damping. As found in the article, 6 OCTOBER 2000 VOL 290 SCIENCE, by Y.Y. Lin, these two nonlinear interactions cause resurrection and rephasing of completely dephased magnetization (the magnetization after a z-gradient).

The images below are the simulation's output of the same situation, and luckily produce the same results as in the paper. They were plotted using the **plottraj** matlab function.



```
#parameter file for 1 pulse - 1 Grad Z sequences
#grid units in cm
dim 1,1, 100
gmin -0.02,-0.02, -0.004693
gmax 0.02, 0.02, 0.004693

#cylinder shape min and max
smin 0,0, -0.004693
smax .003, 6.28, .004693

#fid pieces
npts 512
tf 2
#the pulse bits
pulseangle1 90
pulseamp 80000

# ----- THE input file....
#basic spin parameters
Bo 14.1
```

```

temperature 300
offset 0
T2 0
T1 0
spintype 1H
eps 1e-3

demagOff 0

#95% water (2 protons a pieces)
moles 0.1045

#the extra interactions parts
raddamp 0.01

## #gradient things
#choose 'real gradient'(n) or ideal initial condition(y)
#if ideal magnetization will be spread evenly
#around a circle in the xy plane
ideal y
#non-ideal bits (grad units in Gauss/cm)
grad 0,0,1
gradtime1 0.005

fidout data
magout mag
trajectories traj

-----end input file-----
#include "blochlib.h"

/*
this is an attempt to imitate the result from YY Lin in
6 OCTOBER 2000 VOL 290 SCIENCE
*/
/*
RF ---90x---FID
Grad -----Gzt-----

*/
//need these two namespaces....
using namespace BlochLib;
using namespace std;

timer stopwatch;
void printTime(int nrounds=1){
    std::cout <<std::endl<< "Time taken: " << (stopwatch()/nrounds) << " seconds" << endl;
}

void Info(std::string mess)
{
    std::cout << mess << endl;
    std::cout.flush();
}

//some typedefs to make typing easier
typedef XYZcylinder TheShape;
typedef XYZshape<TheShape> TheGridS;
typedef GradientGrid<TheGridS > TheGrid;
typedef ListBlochParams< TheGrid, BOptions::Particle|BOptions::HighField, double > MyPars;

//Extra ineractions
typedef Interactions<Offset<MyPars>, Relax<>, RadDamp, ModulatedDemagField > MyInteractions;

```

```

//typedefs for Bloch parameter sets
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

int main(int argc,char* argv[]){
    std::string fn;
    query_parameter(argc,argv,1, "Enter file to parse: ", fn);
    Parameters pset(fn);
    double pangl=pset.getParamD("pulseangle1");
    double amp=pset.getParamD("pulseamp");

    int nsteps=pset.getParamI("npts");
    double tf=pset.getParamD("tf");

    std::string fout=pset.getParamS("fidout");
    std::string magout=pset.getParamS("magout");
    //int confid=pset.getParamI("allfid","",false);

    int cv=pset.getParamI("lyps", "", false);
    std::string lypfile=pset.getParamS("lypout", "", false, "lyps");

    std::string dataou=pset.getParamS("trajectories", "", false);

//gradient pars
    double gradtime1=pset.getParamD("gradtime1");      //first grad pulse time

/*****
//Grids
    coord<int> dims(pset.getParamCoordI("dim"));
    coord<> mins(pset.getParamCoordD("gmin"));
    coord<> maxs(pset.getParamCoordD("gmax"));

    coord<> smins(pset.getParamCoordD("smin"));
    coord<> smaxs(pset.getParamCoordD("smax"));

    Info("Creating grid....");
    Grid<UniformGrid> gg(mins, maxs, dims);
    Info("Creating initial shape....");
    TheShape tester(smins, smaxs);
    Info("Creating total shape-grid....");
    TheGridS grids( gg, tester);
    std::ofstream goo("grid");
    goo<<grids<<std::endl;
//create the gradient grids..

    char ideal=pset.getParamC("ideal");
    coord<> grad=pset.getParamCoordD("grad");

    Info("Creating Gradient map grids....");
    TheGrid jj(grids);

    jj.G(grad);
/*****


//set up Parameter lists
    int nsp=jj.size();
    Info("Creating entire spin parameter list for "+itost(nsp)+" spins....");
    MyPars mypars(jj.size(), "1H", jj);
    nsp=mypars.size();

    double inBo=pset.getParamD("Bo");
    double inTemp=pset.getParamD("temperature");
    std::string spintype=pset.getParamS("spintype");
    double moles=pset.getParamD("moles");

```

```

std::string detsp=spintype;

Info("setting spin parameter offsets....");
for(int j=0;j<nsp;j++){
    mypars(j)=spintype;
    mypars(j).Bo(inBo);
    mypars(j).temperature(inTemp);
}

mypars.calcTotalMo();
mypars.print(std::cout);
/****************************************/
//The pulse list for a real pulse on protons..
Info("Creating real pulse lists...
");

Pulse PP1(spintype, amp, 0.); // (spin, amplitude, phase, offset)
PP1.print(std::cout);
double tpulse=PP1.timeForAngle(pang1*Pi/180., spintype);

/****************************************/
//time train
double tct=0;
Info("Initializing Time train for first Pulse....");
TimeTrain<UniformTimeEngine> P1(0., tpulse, 10,100);
tct+=tpulse;
Info("Initializing Time train for First Gradient Pulse....");
TimeTrain<UniformTimeEngine> G1(tct, tct+gradtime1, 50,100);
tct+=gradtime1;
Info("Initializing Time train for FID....");
TimeTrain<UniformTimeEngine> F1(tct, tf+tct, nsteps,5);
if(ideal=='y'){ F1.setBeginTime(0); F1.setEndTime(tf); }

/****************************************/
/****************************************/
//interactions
double t2s=pset.getParamD("T2");
double t1s=pset.getParamD("T1");
double offset=pset.getParamD("offset")*PI2;

//demag field 'time constant'
//because we are in the 'particle' rep
// we need to calculate the real Mo separately
double mo=mypars[0].gamma()*hbar*
    tanh(hbar*PI*(inBo*mypars[0].gamma()/PI2)/kb/inTemp)
    *No*moles*1e6/2.0;
double demag=1.0/(mo*permVac*mypars[0].gamma()));

double tr=pset.getParamD("raddamp");

Info("setting Interactions....");

Offset<MyPars> myOffs(mypars, offset);
Relax<> myRels(mypars, (!t2s)?0.0:1.0/t2s, (!t1s)?0.0:1.0/t1s);
RadDamp RdRun(tr);
ModulatedDemagField DipDip(demag, jj.G());
std::cout<<"Total Mangetization: "<<mo<<std::endl;
std::cout<<DipDip<<" Td: "<<DipDip.td()<<" axis: "<<DipDip.direction()<<std::endl;

MyInteractions MyInts(myOffs, myRels, RdRun, DipDip);
demag=pset.getParamD("demagOff", "", false, 0.0);
if(demag!=0) DipDip.off();

/****************************************/

```

```

//THis is the BLoch solve to perform a pulse

Info("Initializing total parameter list with a pulse....");
PulseBloch myparspulse(mypars, PPl, MyInts);

//This is the Bloch solver to Collect the FID (i.e. has no pulses...FASTER)
Info("Initializing total parameter list for FID collection....");
NoPulseBloch me;
me=myparspulse;

Vector<coord<> > tm=me.currentMag();
std::cout<<"TOTAL mag initial condition: "<<sum(tm)<<std::endl;

//the 'error' in the helix
double emp=pset.getParamD("eps", "", false, 1e-3);

//set the circular initial condition..a single helix
if(ideal=='y'){
    MyPars::iterator myit(mypars);
    double lmax=smaxs.z()-smins.z();
    coord<> tp;
    while(myit){
        tp=myit.Point();
        tm[myit.curpos()].x()=sin(tp.z()/lmax*PI2)+emp;
        tm[myit.curpos()].y()=cos(tp.z()/lmax*PI2);
        tm[myit.curpos()].z()=0.0;
        ++myit;
    }
}
stopwatch.reset();

BlochSolver<PulseBloch > drivP(myparspulse, tm);
BlochSolver<NoPulseBloch > drivD(me, tm);

//integrate pulse and gradient pulse
//only if NOT ideal experiment
if(ideal=='n'){
    //output trajectory data if wanted
    if(dataou!=""){
        drivP.setWritePolicy(Continous);
        drivP.setRawOut(dataou, std::ios::out);
    }else{
        drivP.setWritePolicy(Hold);
    }
    drivP.setCollectionPolicy(FinalPoint);

//integrate the first pulse
myOffs.off();      //turn off gradient
Info("Integrating first Pulse ....");

if(!drivP.solve(P1)){
    Info(" ERROR!!! could not integrate pulse P1....");
    return -1;
}

//integrate the gradient pulse

Info("Integrating the Gradient Pulse....");
drivD.setInitialCondition(drivP.lastPoint());
//output trajectory data if wanted
if(dataou!=""){
    drivD.setWritePolicy(Continous);
    drivD.setRawOut(dataou, std::ios::app|std::ios::out);
}else{
    drivD.setWritePolicy(Hold);
}

```

```

    }

    if(gradtime1>0){
        myOffs.on();      //turn on gradient
        if(!drivD.solve(G1)){
            Info(" ERROR!!!..could not integrate G1....");
            return -1;
        }
    }
}

//integrate FID
if(cv){
    me.calcVariational();
    drivD.setVariationalInitCond(me.curVariational());
    drivD.setLyapunovPolicy(LypContinous);
    drivD.setLypDataFile(lypfile);
}

myOffs.off();
Info(
Integrating for FID ....
");

//output trajectory data if wanted
drivD.setCollectionPolicy(MagAndFID);
if(dataou!=""){
    drivD.setWritePolicy(Continous);
    if(ideal=='y') drivD.setRawOut(dataou, std::ios::out);
    else drivD.setRawOut(dataou, std::ios::app|std::ios::out);
}else{
    drivD.setWritePolicy(Hold);
}

//solve the FID and write it to a file
if(drivD.solve(F1)){
    drivD.writeSpectrum(fout);
    drivD.writeMag(magout);
}

printTime();
}

```

--Constructors

--- Bloch

--- Bloch

--- Bloch

--- Bloch

--- Bloch

--- Bloch

--Element Extraction

--- interactions

--- parameters

--- pulses

--Assignments

--- operator=

--- setInteractions

--- setParameters

--- setPulses

--Other Functions

--- calcVariational

--- currentMag

--- curVariational

--- initialCondition

--- noCalcVariational

--- setIntialCondition

--- size()

--- variationalPolicy

```
template<class Parameters_T, class Pulse_T, class Interactions_T>
class Bloch {....}
```

This class is designed to be the master function object to the ODE solvers and the BlochSolver. It wraps the ListBlochParameters, Pulse (or NoPulse) and the interactions object to specifically optimize the 'void function(double time, Vector<coord> &y, Vector<coord> &dydt)' functional call based on the parameters, the Offset_T of the parameters, pulseing or no pulses, and the interaction set.

Parameters_T --> is (in every case) some ListBlochParameter class definition.

Pulse_T --> is either 'Pulse' if there is a pulse on the system and 'NoPulse' if there is no pulse on the system.

Interaction_T--> is an Interaction class.

This classes main interface is for you to set the initial conditions, parameters, and pulses of the class (the 'function' interface you will probably never use). It give you the ability to easily specify weather or not to calculate the variational equations (use to calculate Lyapunov numbers).

The 'function' objects are of the same for as required by the ODE solver (and not described below). There are also some 'helper' functions to aid in preparation and excecution of the 'function' that are also not described below.

For context usage see the examples in 'DemagField', 'DipoleDipole', 'Relax', 'BulkSus', and 'RadDamp'.

Bloch::constructor

Function: **Bloch()**;

Input: void

Description: Creates a Bloch an empty object. Not very usful, as you will need to set the parameters, pulses, and interactions later.

Example Bloch myBloch;

Usage:

Bloch::constructor

Function: `bloch(Parameters_T &pars)`

Input: `pars` --> An input Parameters set (a ListBlochParam<Grid_T, BOptions, Offset_T>)

Description: `Set up a basic bloch object using the input parameters 'pars'. If you have a Pulse and Interactions then you must set those later. It maintains a pointer to the Parameters object, so if you delete that parameters object, the Bloch will probably crash the program if used...`

Example `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj(gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars parsIn(nsp, "1H", jj);

//need to typedef the MyInteractions
//use 'Pulse' or 'NoPulse' for the Pulse_T
typedef Bloch< MyPars, Pulse_T, MyInteraction> PulseBloch;

//sets the parameters pointer, however,
// you may need to set the Pulse pointer
// (if Pulse_T != NoPulse)
//and the set the interaction pointers
PulseBloch pbloch(pars);
```

Bloch::constructor

Function: `■ BLoch(Parameters_T &in, Pulse &pul);`

Input: `■ pars --> An input Parameters set (a ListBlochParam<Grid_T, BOptions, Offset_T>)`
`■ pul --> A pulse object`

Description: `■ The basic constructor for No Interactions and a Pulse. It sets the Parameters_T (a ListBLochParam) pointer and the Pulse pointer. If you destroy the Pulse or the Parameter_T object and use the Bloch object, more then likely the program will crash (becuase now the pointers are undefined).`

Example `■ typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Bloch< MyPars, Pulse, NoInteractions> PulseBloch;

// (spin, amplitude, phase)
Pulse PP1("1H", 150000, 0.);

//sets the parameters pointer, however,
// and the Pulse pointer
PulseBloch pbloch(pars, PP1);
```

Bloch::constructor

Function: □ **Bloch(Parameters_T &pars, Pulse &pul, Interaction_t &inter):**

Input: □ pars --> An input Parameters set (a ListBlochParam<Grid_T, BOptions, Offset_T>)
pul --> A pulse object
inter--> An Interactions object

Description: □ The master constructor with Pulse_T != NoPulse Interactions_T != NoInteractions. Creates an object using all three basic template arguments. If you destroy the Pulse, Parameter_T, or the Interactions_T object and use the Bloch object, more then likely the program will crash (because now the pointers are undefined).

Example □ `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

// (spin, amplitude, phase)
Pulse PP1("1H", 150000, 0.);

typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;

//sets the parameters pointer,
// and the Pulse pointer
// and the Interactions pointer
PulseBloch pbloch(pars, PP1, inters);
```

Bloch::constructor

Function: □ **Bloch(Parameter_T &pars, Interactions_T &inter)**

Input: □ pars --> An input Parameters set (a ListBlochParam<Grid_T, BOptions, Offset_T>) inter--> An Interactions object

Description: □ The basic constructor for Pulse_T=NoPulse and an Interactions object. It sets the Parameters_T (a ListBLochParam) pointer and the Interactions_T pointer. If you destroy the Interactions_T or the Parameter_T object and use the Bloch object, more then likely the program will crash (becuase now the pointers are undefined).

Example □ `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//sets the parameters pointer, however,
// and the Interactions pointer
NoPulseBloch pbloch(pars, inters);
```

Bloch::constructor

Function: □ `Bloch<Parameters_T, Pulse_T, Interactions_T> &cp)`

Input: □ cp --> another Bloch object

Description: □ Copies an existing Bloch object to another. Some rules applied to the template arguments.

The Parameters_T must be the same for BOTH the new object and the copied object.

If Pulse_T for the new object = NoPulse then Pulse_T of the copied object can be either Pulse or NoPulse (the Pulse will not be copied).

If Pulse_T for the new object = Pulse then Pulse_T of the copied object can be either Pulse or NoPulse. If the copied object has Pulse_T=NoPulse, then NO pulse will be copied and you will need to define it later.

If Interactions_T for the new object = NoInteractions then no Interactions will be copied from the copied object.

If Interactions_T for the new object != NoInteractions, then BOTH the Interactions_T for the two objects MUST BE THE SAME.

Example □ `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BPOptions::Particle | BPOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;

//sets the parameters pointer
// and the Interactions pointer
NoPulseBloch npbloch(pars, inters);

//now we need to set thePulse pointer
PulseBloch pbloch(npbloch);
Pulse PP1("1H", 150000, 0.);
pbloch.setPulses(PP1);

//a new NoPulse object
NoPulseBloch npbloch2(npbloch);
```

```
//can also make one like this  
// the pulse is ignored  
NoPulseBloch npbloch3(pbloch);
```

Bloch::element extraction

Function: □ **Interactions_T *interactions();**

Input: □ void

Description: □ Returns the pointer to the current Bloch interaction object.

Example Usage: □

```
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrid;

Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BPOptions::Particle | BPOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//sets the parameters pointer
NoPulseBloch npbloch(pars);

// and the Interactions pointer
npbloch.setInteractions(inters);

MyInteractions newguy;
newguy=( *npbloch.interactions());
```

Bloch::element extraction

Function: **Parameters_T *parameters();**

Input: void

Description: Returns the pointer to the current Parameter_T object in Bloch.

Example Usage:

```
typedef XYZfull TheShape;
typedef XYZshape<TheShape> TheGrid;

Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BPOptions::Particle | BPOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;
NoPulseBloch npbloch(pars);

MyPars pars2;
pars2=(*npbloch.parameters());
```

Bloch::element extraction

Function: **Pulse *pulses();**

Input: void

Description: Returns the pointer to the current Bloch Pulse object.

Example Usage:

```
typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;
Pulse PP1("1H", 82000, 0.);
PulseBloch pbloch(pars, PP1, inters);
Pulse PP2("1H", 150000, 0.);
pbloch.setPulses(PP1);

Pulse PP3;
PP3=(*pbloch.pulses());
```

Bloch::assignments

Function: □ `Bloch operator=(Bloch<Parameters_T, Pulse_T, Interactions_T> &cp)`

Input: □ cp --> another Bloch object

Description: □ Copies an existing Bloch object to another. Some rules applied to the template arguments.

The Parameters_T must be the same for BOTH the new object and the copied object.

If Pulse_T for the new object = NoPulse then Pulse_T of the copied object can be either Pulse or NoPulse (the Pulse will not be copied).

If Pulse_T for the new object = Pulse then Pulse_T of the copied object can be either Pulse or NoPulse. If the copied object has Pulse_T=NoPulse, then NO pulse will be copied and you will need to define it later.

If Interactions_T for the new object = NoInteractions then no Interactions will be copied from the copied object.

If Interactions_T for the new object != NoInteractions, the BOTH the Interactions_T for the two objects MUST BE THE SAME.

Example □ `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;

//sets the parameters pointer
// and the Interactions pointer
NoPulseBloch npbloch(pars, inters);

//now we need to set the Pulse pointer
PulseBloch pbloch=npbloch;
Pulse PP1("1H", 150000, 0.);
pbloch.setPulses(PP1);

//a new NoPulse object
```

```
NoPulseBloch npbloch2=npbloch;  
//can also make one like this  
// the pulse is ignored  
NoPulseBloch npbloch3=pbloch;
```

Bloch::assignments

Function: **void setInteractions(Interactions_T &in);**
 void setInteractions(Interactions_T *in);

Input: in --> an Interactions object.

Description: Sets the Interaction pointer to the input. The Interactions_T must be the same as the one initially declared in Bloch.

Example **typedef XYZfull TheShape;**

Usage: **typedef XYZshape<TheShape> TheGrid;**

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Interactions<Offset<> > MyInteractions;

Offset offs(pars, 2000.0*PI2);
MyInteractions inters(offs);

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

//sets the parameters pointer
NoPulseBloch npbloch(pars);

// and the Interactions pointer
npbloch.setInteractions(inters);
```

Bloch::assignments

Function: **void setParameters(Parameters_T &in);**
 void setParameters(Parameters_T *in);

Input: in --> a set of parameters

Description: Set the parameters point inside the Bloch object. It MUST be the same class type as the initially declared in Bloch. NOTE:: this function will REINITIALIZE the initial condition (because the size of the parameters can change). So should you desire to keep the same initial condition, but change the parameters, you should first get the current Magnetization vector (via currentMag()), set the Parameters, then reset the current Magnetization vector (assuming that the size of the parameters is the same size as the old magnetization vector) (via setCurrentMag()) back to the old one.

Example **typedef XYZfull TheShape;**

Usage: **typedef XYZshape<TheShape> TheGrid;**

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;
NoPulseBloch npbloch(pars);

//make a new parameter set
MyPars pars2(nsp, "13C", jj);

//get the old initial magnetization
Vecotor<coord<> > tmM=npbloch.currentMag();

//set the new parameters
npbloch.setParametes(pars2);

//reset the magnetization
npbloch.setCurrentMag(tmM);
```

Bloch::assignments

Function: `void setPulses(Pulse &in);`
`void setPulses(Pulse *in);`

Input: `in` --> a Pulse object (or pointer to a Pulse object)

Description: `Sets the Pulse pointer for a Bloch Object with Pulse_T!=NoPulse.`

Example Usage: `typedef Bloch< MyPars, Pulse, MyInteractions> PulseBloch;`
`Pulse PP1("1H", 82000, 0.);`
`PulseBloch pbloch(pars, PP1, inters);`
`Pulse PP2("1H", 150000, 0.);`
`pbloch.setPulses(PP1);`

Bloch::other

Function: □ void calcVariational()

Input: □ void

Description: □ This tells the object that along with the normally evolved magnetization, the Variational equations will also be evolved. For each 3-vector magnetization, there are 3 3-vector variational equations to calculate (really a 3x3 matrix) (thus the ODE solver sees 4*Number of Spins) equations and the time evolution will take 3 times as long. The initial condition for the variational equations is always the identity matrix for each spin. This function also sets the initial condition.

Example

□

```
//define your parameters....  
//...  
//  
  
typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;  
  
NoPulseBloch npbloch(pars);  
npbloch.calcVariational();
```

Usage:

Bloch::other

Function: `□ Vector<coord<> > currentMag()`

Input: `□ void`

Description: `□ Returns the current magntization. It is initially the 'initialCondition().'`

Example Usage: `□ //define your parameters and interactions
//...

typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;`

```
NoPulseBloch npbloch(pars);  
Vector<coord<> > curM=npbloch.currentMag();
```

Bloch::other

Function: `Vector<coord> > curVariational()`

Input: `void`

Description: `>Returns the current variational evolution points(if the 'calcVariational()' function has been called, otherwise a warning will be issued). The output vector will be 3*number-of-spins.`

Example `//define your parameters and interactions`

Usage: `//...`

```
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

NoPulseBloch npbloch(pars);
npbloch.calcVariational();

Vector<coord> > curV=npbloch.curVariational();
```

Bloch::other

Function: `Vector<coord<> > initialCondition()`

Input: void

Description: Returns the initial condition assigned to the Bloch object. This will be set upon the calling of the constructor or 'setInitialCondition()'. This does NOT evolve with the ODE solver, so you can 'remember' what it was.

Example

Usage: `//define your parameters....`

`//...`

`//`

```
typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;
```

```
NoPulseBloch npbloch(pars);
```

```
//will vary depending
```

```
// on what you set the 'InitialCondition'
```

```
// functional form
```

```
// in ListBLochParams to be
```

```
Vector<coord<> > m0=npbloch.initialCondition();
```

Bloch::other

Function: `void noCalcVariational()`

Input: `void`

Description: `Will set the Magnetization back to a size that is the number of spins (if the 'calcVariational()' was called. It does nothing if variationalPolicy()==NoVariational.`

Example `//define your parameters....`

Usage: `//...
//`

```
typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;  
  
NoPulseBloch npbloch(pars);  
npbloch.calcVariational();  
  
//reste back to the nonvariation version  
npbloch.noCalcVariational();
```

Bloch::other

Function: `void setInitialCondition(const Vector<coord> &in);`

Input: `in` --> the new initial condition

Description: `in` Sets the initial condition in the Bloch Object. It will overwrite both the time evolved Magnetization vector AND the initial condition vector.

Example `//define your parameters....`

Usage: `//...
//`

```
typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;  
  
NoPulseBloch npbloch(pars);  
  
//will vary depending  
//on what you set the 'InitialCondition'  
//functional form  
//in ListBlochParams to be  
Vector<coord> m0=npbloch.initialCondition();  
  
m0+=5;  
npbloch.setInitialCondition(m0);
```

Bloch::other

Function: **int size();**

Input: void

Description: Returns the size of the current spin space...this is NOT the size of the magnetization vector (that can be size() or 4*size() depending on whether or not you are calculating the variation equations as well).

Example `typedef XYZfull TheShape;`

Usage: `typedef XYZshape<TheShape> TheGrid;`

```
Grid<UniformGrid> gg(mins, maxs, dims);
TheShape tester;
TheGrid jj( gg, tester);

typedef ListBlochParams<
TheGrid,
BOptions::Particle | BOptions::HighField,
double > MyPars;
MyPars pars(nsp, "1H", jj);

typedef Bloch< MyPars, NoPulse, NoInteractions> NoPulseBloch;
NoPulseBloch npbloch(pars);

cout<<mpbloch.size();
```

Bloch::other

Function: **BlochOps::VariationEqs variationalPolicy()**

Input: void

Description: Returns an enum value either {BlochOps::Variational, BlochOps::NoVariational} depending on whether or not you used the 'calcVariational()' function.

Example //define your parameters and interactions

Usage: //...

```
typedef Bloch< MyPars, NoPulse, MyInteractions> NoPulseBloch;

NoPulseBloch npbloch(pars);
npbloch.calcVariational();

if(npbloch.variationalPolicy()==BlochOps::Variational){
cout<<"Calculating variational Eqs"<<endl;
}
```

--Constructors template<class Bloch_T, class ODEs_T=bs<Bloch_T, coord<>, Vector<coord<> >>
--- BlochSolver
--- BlochSolver

--Element Extraction

--- FID

--- lastPoint

--- M

--Assignments

--- setBloch

--- setCollectionPolicy

--- setDetect

--- setDetect

--- setInitialCondition

--- setLyapunovPolicy

--- setProgressBar

--- setRawOut

--- setWritePolicy

--Other Functions

--- solve

This class adds a nice interface between the ODE solver and user, specifically for the NMR type experiment. It, by default, uses the 'bs' ODE integrator (the best one for the job in most any NMR case). It allows the user a variety of data collection, display, and data output possibilities from the data generated from the ODE solver.

Its main focus is as a memory saver. If one was to simulate 1000 spins over 1024 data points and save the entire data chunk in memory, not only would performance be affected, but the memory requirements for 3000*1024 doubles is quite large. So it provides a seamless way to continuously dump data to a file without holding it in memory. It also provides simple 'detection' methods for collection of the magnetization and an 'FID' for the desired spins in the entire list. It can also save a bit of initialization of the ODE step size determination by carrying the ODE object around. Finally the progress of the integrator can be displayed to the console (or not displayed) with a simple 'switch.'

The main interface provides methods for setting initial conditions and simple 'policy' settings for data collection and output.

For context usage see the examples in 'DemagField', 'DipoleDipole', 'Relax', 'BulkSus', and 'RadDamp'

BlochSolver::constructor

Function: **BlochSolver<Bloch_T>()**

Input: void

Description: The empty constructor. Sets the default policies the Bloch pointer is NULL and the intial condition is empty.

Example `typedef Bloch<Parameter_T, Pulse_T, Interactions_T> mybloch;`

Usage:
`BlochSolver<mybloch> emptybs;`

BlochSolver::constructor

Function: `BlochSolver<Bloch_T>(Bloch_T &bloch, const Vector<coord> > &initC);`

Input: bloch --> A Bloch object (the main function)
 initC --> the initial condition

Description: The basic constructor. Sets the Bloch_T pointer to the input (NOTE: if the Bloch_T object is destroyed before the BlochSolver, and the Solver is used, the program will likely crash). The Initial Condition to the solver is also set to initC.

Example `typedef Bloch<Parameter_T, Pulse_T, Interactions_T> mybloch;`
Usage: `mybloch bloch(pars, pulse, inters);`

```
BlochSolver<mybloch> bsolver(bloch, bloch.currentMag());
```

BlochSolver::element extraction

Function: `Vector<complex> FID();`

Input: `void`

Description: `>Returns the FID calculated fom the solver. It returns a non-empty vector when the collection policy is SolverOps::FID,SolverOps::MagAndFID, or SolverOps::All.`

Example `BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continous);`

```
bsolver.setDetect("1H");

//solver things...

Vector<complex> fdi=bsolver.FID();
```

BlochSolver::element extraction

Function: `■ Vector<coord> > lastPoint();`

Input: `■ void`

Description: `■` Retrieves the last integrated point performed by the solver. It will contain both the Mangetization points and the Variational data (if the Variational points were desired)

Example `■ BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continous);`

```
std::string trajName="traj.out";
//the initall opeing of the file
bsolver.setRawOut(trajName);

//solve things

//set up a new solver using the last point
//of the previous integration as the
//starting point for this one
BlochSolver<AntoherBloch> bsolver2(mybloch2, bsolver.lastPoint());

//dump all the data
bsolver2.setWritePolicy(SolverOps::Continous);

//wish to appen the new data
// to the end of the same file
bsolver.setRawOut(trajName, std::ios::app|std::ios::out);
```

BlochSolver::element extraction

Function: `Vector<coord<> > M();`

Input: void

Description: Returns the magnetization vector. It will return a non-empty vector when the collection policy is non-empty vector when the collection policy is SolverOps::Magnetization, SolverOps::MagAndFID, or SolverOps::All.

Example `BlochSolver<MyBloch> bsolver(mybloch);`
Usage: `bsolver.setWritePolicy(SolverOps::Continuous);`

`bsolver.setDetect("1H");`

`//solve things...`

`Vector<coord<> > mag=bsolver.M();`

BlochSolver::assignments

Function: `void setBloch(Bloch<...> &inb);`

Input: `inb-->` a Bloch object

Description: `inb-->` Sets a new Bloch function object. It must be of the same type as the template argument. BlochSolver maintains a Pointer to this object, so if the Bloch object is destroyed before the Solver, most likely the program will crash of teh Solver object is used.

Example `BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continous);`

```
std::string trajName="traj.out";
//the initall opeing of the file
bsolver.setRawOut(trajName);

//solve things

//set up a new solver using the last point
//of the previous integration as the
//starting point for this one
BlochSolver<AntoherBloch> bsolver2;
bsolver2.setBloch(bloch2);
```

BlochSolver::assignments

Function: **void setCollectionPolicy(SolverOps::collection colpol);**

Input: colpol --> a SolverOps collection policy (All,Magnetization, MagAndFID, FIDonly, FinalPoint)

Description: Sets the data collection policy for each time integrated step. The available policies are...

SolverOps::All --> collect ALL the data (NOTE:: this can be quite memory intensive and thus slow the program down, I do not recommend using this, instead use the WritePolicy,
SolverOps::Continuous to dump the trajectories as you go (takes no memory to do this)).

SolverOps::Magnetization --> collect only the TOTAL magnetization for the spins you wish to detect (see 'setDetect'). This is Vector<coord> object that store Mx, My, Mz as a function of time.

SolverOps::MagAndFID --> will collect both the TOTAL magnetization and the TOTAL FID (simply $M_x + i M_y$) for the detected spins (see 'setDetect').

SolverOps::FIDonly--> will collect only the TOTAL FID ($M_x + i M_y$) for the detected spins (see 'setDetect').

SolverOps::FinalPoint --> store ONLY the final integrated point from the solver. This useful for things like 90 pulses when you will simply pass this last point to the next integrator.

the default is SolverOps::MagAndFID.

Example `//define the Bloch object...`

Usage:

```
BlochSolver<MyBloch> bsolver(mybloch);
bsolver.setCollectionPolicy(SolverOps::MagAndFID);
```

BlochSolver::assignments

Function: `void setDetect(std::string spinLabel);`

Input: `spinLabel` --> a spin isotope label (like "1H", "13C", etc).

Description: `Set the isotope to 'detect.'` It is used for collection of the FID and Magnetization. If it is not set, then all spins will be detected if the collection policy is set to SolverOps::MagAndFID,SolverOps::Magnetization, SolverOps::FID, or SolverOps::All.

Example `BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continous);`

```
bsolver.setDetect("1H");
```

BlochSolver::assignments

Function: **void setDetect(const Vector<int> &list);**
 void setDetect(const Range &list);

Input: list --> a list of integers specifying which spins in the parameter list should be detected.

Description: Sets the spins that will be detected from the indexes stored in the input list. It is used for collection of the FID and Magnetization. If it is not set, then all spins will be detected if the collection policy is set to SolverOps::MagAndFID,SolverOps::Magnetization, SolverOps::FID, or SolverOps::All.

When use the Range object ...DO NOT use Range::Start and Range::End as the end nor start is specified anywhere.

Example BlochSolver<MyBloch> bsolver(mybloch);
Usage: bsolver.setCollectionPolict(SolverOps::FID);

`//collect every other spin
Vector<int> list(mypars.size());
list=Range(0, 10, Range::End);
bsolver.setDetect(list);`

BlochSolver::assignments

Function: `void setInitialCondition(const Vector<coord> &InitialCon)`

Input: `InitialCon` --> the new initial condition

Description: `Sets the initial condition for the solver. It DOES NOT overwrite the initial condition set in the Bloch object. The two are separated from each other. It should be os the same length as the number of spins. If the Variational equations are desired then it should be of length 4*num-of-spins.`

Example `BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continous);`

```
std::string trajName="traj.out";
//the initall opeing of the file
bsolver.setRawOut(trajName);

//solve things

//set up a new solver using the last point
//of the previous integration as the
//starting point for this one
BlochSolver<AntoherBloch> bsolver2;
bsolver2.setBloch(bloch2);
bsolver2.setInitialCondition(bsolver.lastPoint());
```

BlochSolver::assignments

Function: `void setLyapunovPolicy(SolverOps::lyapunovcalc lypol);`

Input: `lypol` --> a SolverOps lyp policy (SolverOps::LypNoCalc, SolverOps::LypHold, or SolverOps::LypContinous)

Description: `l` Sets the collection and calulation of Lyapunov exponent calculation and data collection. You must set the 'calcVariational()' in you Bloch in order for these to work. If you turn on the policy without setting the Variational data set it will AUTOMATICALLY call 'calcVariational()' in the Bloch object as well as issuing a warning. The available policies are

SolverOps::LypNoCalc --> do not calcualte them or store anything at all

SolverOps::LypContinous --> Calculate them, but as soon as they are calculated dump them to a file. This is recommended as for each spin there will be 3 exponents (one for Mx, My, and Mz) thus the data storage for these can be enormous for large spin sets. If you use this one, you should also set the output file name with 'setLypDataFile()' otherwise it will dump the exponenets to a file named 'Lyapunov.out'.

SolverOps::LypHold --> Calculate them, and store them. This is NOT recommended becuase the memmory requirements for them can be huge.

the default is SolverOps::LypNoCalc.

Example `//define the Bloch object...`

Usage:

```
BlochSolver<MyBloch> bsolver(mybloch);
bsolver.setWritePolicy(SolverOps::Continous);

mybloch.calcVariational();
bsolver.setLyapunovPolicy(SolverOps::LypContinous);
bsolver.setLypDataFile("mylyps");
```

BlochSolver::assignments

Function: `void setProgressBar(SolverOps::progress progp1);`

Input: `progp1` --> the Solver Ops progress policy (On or Off)

Description: `This` sets weather or not to display the progress bar to the console. It is simply a little visual aid to show you where in the integration the solver is...for 'fast' integrations (like 1-2 spins) the progress bar can actually hurt performance (i.e. it takes almost as much processor to display the rapidly updating bar as it does to do the integration)...also if you plan on 'backgrounding' the program, it is advisable to turn it off as it is useless and can potentially interfere with your terminal.

The available options are..

`SolverOps::On` --> turn on the progress bar

`SolverOps::Off` --> turn off the progress bar

default is `SolverOps::On`

Example `//define the Bloch object...`

Usage:

```
BlochSolver<MyBloch> bsolver(mybloch);
bsolver.setProgressBar(SolverOps::Off);
```

BlochSolver::assignments

Function: **void setRawOut(std::string inname, std::ios::openmode iosf=std::ios::out);**

Input: inname --> the output filename
iosf --> the open mode for the file (should be ios::out or std::ios::app|std::ios::out)

Description: Sets the raw data ouput file. This file is used IF the Write policy is 'SolverOps::Continous'. If you desire to collect tragectories between different BLochSolver objects, then set the openmode to 'std::ios::app|std::ios::out' to append the new data to the end of the file. The first call to this function, however, should simply be 'ios::out' as one probably wishes to write over the old data.

Example BlochSolver<MyBloch> bsolver(mybloch);
Usage: bsolver.setWritePolicy(SolverOps::Continous);

```
std::string trajName="traj.out";
//the initall opeing of the file
bsolver.setRawOut(trajName);

//solve things

//set up a new solver using the last point
//of the previous integration as the
//starting point for this one
BlochSolver<AntoherBloch> bsolver2(mybloch2, bsolver.lastPoint());

//dump all the data
bsolver2.setWritePolicy(SolverOps::Continous);

//wish to appen the new data
// to the end of the same file
bsolver.setRawOut(trajName, std::ios::app|std::ios::out);
```

BlochSolver::assignments

Function: `void setWritePolicy(SolverOps::writePol wpol);`

Input: `wpol` --> A SolverOps flag (SolverOps::Hold, SolverOps::Continous)

Description: `This allows you to set simple data writing polices as the solver integrates. The allowed 'writePol' are`

`SolverOps::Hold --> write nothing`

`SolverOps::Continous --> write the trajectories as each time step is encountered in the ODE solver
(allows one to see the data before the integrator is finished). It dumps out ALL the trajectory data
except the Variational elements.`

`The default is SolverOps::Hold`

Example `//define the Bloch object...`

Usage:

```
BlochSolver<MyBloch> bsolver(mybloch);
bsolver.setWritePolicy(SolverOps::Continous);
```

BlochSolver::other

Function: `bool solve(TimeTrain<TimerEng> &therun);`

Input: `therun` --> a time train specifying the collection points, times, and substeps...

Description: `The master solver. It returns true if everything was solved happily, and false if it failed. It performs the integration, Lyapunov calculations (if desired), and the various data outputting (if desired).`

Example `BlochSolver<MyBloch> bsolver(mybloch);`

Usage: `bsolver.setWritePolicy(SolverOps::Continuous);`

`bsolve.setCollectionPolicy(SolverOps::MagAndFID);`

```
std::string trajName="traj.out";
//the initial opening of the file
bsolver.setRawOut(trajName);
```

```
//a time train
TimeTrain<UniformTimeEngine> myT(0, 2.0, 1024,1);
```

```
//solve it
bsolver.solve(myT);
```

```
//dump the data to the console
bsolve.FID().print(cout, " ");
```

```
bsolv
```

--Constructors**--- SpinSys****--- SpinSys****--- SpinSys****--Element Extraction****--- element****--- gamma****--- gammaGauss****--- HS****--- HS****--- mass****--- mass****--- momentum****--- momentum****--- name****--- number****--- qn****--- qn****--- relativeFrequency****--- reseptivity****--- symbol****--- weight****--- weight****--Assignments****--- operator=****--- operator=****--- operator[i]****--Math****--- ==, !=****--IO****--- operator<<****--Other Functions****--- Fp, Fmi,****Fz, Fy, Fx,****F0, Fe****--- heteronuclear****--- homonuclear****--- Ip, Imi,****Iz, Iy, Ix,****Ie****--- isotopes****--- isotopes****Examples****--- 1) simple sys**

class SpinSys:

public spin_sysPars{....

-
- This class acts as a handy container for all the various bits of data associated with LISTS of atomic SpinSys data...(masses, atomic numbers, lables, etc) AND it also contains the Lists of spin operators...
 - This is the Master extension to both "SpinSys" and "MultiSpinOp" and should be used as the interface to them when dealing with 'liquid' like spins systems...
 - *****NOTE***::**ALL The Spin operators ARE generated ONCE upon the the first call to the spin operator (or all are generated upon the 'GenSpinOps' call)...If you have ~15 spins all the generated matrices consume at LEAST (assumeing 15 spin 1/2) ~300Mb of memory...So BE CAREFUL...if you only want a 'few' spin operators uses the generators from the 'SpinOperators' section..see the example below
 - This is done because For solid state simulations most ALL the spin operators are used and used MANY times (due to powder averaging, and sample spinning etc)..to have all the spin-operators 'pre-generated' is a valuable speed enhancment.
 - Much of the Function Nameing was taken from 'Gamma':: this is simply to allow easy transistions..
-

SpinSys::constructor

Function:

SpinSys()

Input:

void

Description:

empty constructor-->NO Spins...no nothing

Example Usage:

SpinSys moo;

SpinSys::constructor

Function:

□ **SpinSys(int nsp)**

Input:

□ nsp--> the number of spins in the system

Description:

□ Creates an SpinSys data set with 'nsp' spins.

Example Usage:

□ `SpinSys moo(4);
int sys=moo.size(); //returns "4"`

SpinSys::constructor

Function:	<code>SpinSys(const SpinSys &cp)</code>
Input:	<code>cp--></code> an existing SpinSys to copy..
Description:	Creates a copy of the old SpinSys. It also COPIES the SPIN-OPS.
Example Usage:	<code>SpinSys moo(3); moo(2) = "14N"; SpinSys loo(moo); //moo is copied into loo</code>

SpinSys::element extraction

Function: `□ std::string element(int which)`

Input: `□ which--> The spin you wish to extract`

Description: `□ Returns the element label of spin 'which.'`

Example Usage: `□ SpinSys moo(3);
cout<<moo.element(2); //prints out 'H'`

SpinSys::element extraction

Function: `double gamma(int which)`

Input: `which-->` The spin you wish to extract

Description: `which` Returns the gamma factor (gyromagnetic ratio) for the nucleous in units of Hz(rad)/Telsa. To get gamma factor in Hz you need to divide this number by 2π

Example `SpinSys moo(2);`

Usage: `cout<<moo.gamma(0); //prints out '26.7519E+7'`

SpinSys::element extraction

Function: `double gammaGauss(int which)`

Input: `which`--> The spin you wish to extract

Description: `which` Returns the gamma factor (gyromagnetic ratio) for the nucleous in units of Hz(rad)/Gauss. To get gamma factor in Hz you need to divide this number by 2Pi

Example `SpinSys moo(2);`

Usage: `cout<<moo.gammaGauss(1); //prints out '26.7519E+3'`

SpinSys::element extraction

Function: `int HS(int which)`

Input: `which`--> The spin you wish to extract

Description: `HS` Returns the hilbert space dimension for the spin ($HS=2*\{\text{spin quantum number}\}+1$) FOR THE SPIN 'WHICH'

This number is essentially the array size needed to accurately describe the spins behavior.

Example `SpinSys moo(4);`

Usage: `cout<<moo.HS(3); //prints out '2'`

SpinSys::element extraction

Function: **int HS()**

Input: none

Description: Returns the hilbert space dimension for the spin (HS=2*<spin quantum number>+1)FOR THE TOTAL SPIN SYSTEM

The number is essentially the array size needed to accurately describe the collective spins behavior.

Example SpinSys moo(4);

Usage: cout<<moo.HS(); //prints out '16'

SpinSys::element extraction

Function:

□ **double mass(int which)**

Input:

□ which--> The spin you wish to extract

Description:

□ Returns the mass in atomic Mass Units (amu) of spin 'which.'

Example Usage:

```
□ SpinSys moo(6);  
moo(4)="27Al";  
cout<<moo.mass(4); //prints out '27'
```

SpinSys::element extraction

Function:

□ **double mass()**

Input:

□ void

Description:

□ Returns the mass in atomic Mass Units (amu) of the TOTAL system.

Example Usage:

```
□ SpinSys moo( 6 );
    moo( 4 ) = "27Al";
    cout << moo.mass(); //prints out '32'
```

SpinSys::element extraction

Function: **std::string momentum(int which)**

Input: which--> The spin you wish to extract

Description: Returns the string form of 'qn(which)'

Example Usage:

```
SpinSys moo(2);  
cout<<moo.momentum(0); //prints out '1/2'
```

SpinSys::element extraction

Function: **std::string momentum()**

Input: void

Description: Returns the string form of 'qn()' (total MAX Momentum).

Example Usage:

```
SpinSys moo(2);  
cout<<moo.momentum(); //prints out '1'
```

SpinSys::element extraction

Function:

□ **std::string name(int which)**

Input:

□ which--> The spin you wish to extract

Description:

□ Returns the full name of the spin which.

Example Usage:

```
□ SpinSys moo(4);  
cout<<moo.name(2); //prints out 'Hydrogen'
```

SpinSys::element extraction

Function:

□ **int number(int which)**

Input:

□ which--> The spin you wish to extract

Description:

□ Returns the atomic number of the spin 'which.'

Example Usage:

□ `SpinSys moo(4);
cout<<moo.number(2); //prints out '1'`

SpinSys::element extraction

Function:

□ **double qn(int which)**

Input:

□ which--> The spin you wish to extract

Description:

□ Returns the spin quantum number of the spin 'which.'

Example Usage:

```
□ SpinSys moo( 3 );
cout<<moo.qn(1); //prints out '0.5'
```

SpinSys::element extraction

Function:

□ **double qn()**

Input:

□ none

Description:

□ Returns the spin quantum number of the TOTAL system.

Example Usage:

□ `SpinSys moo(4);
cout<<moo.qn(); //prints out '2'`

SpinSys::element extraction

Function: **double relativeFrequency(int which)**

Input: which--> The spin you wish to extract

Description: Returns the relative frequency of the spin REALTIVE TO 1H.

Example Usage: SpinSys moo(2);
moo(1)="13C";
cout<<moo.relativeFrequency(0); //prints out '1'
cout<<moo.relativeFrequency(1); //prints out '0.251451'

SpinSys::element extraction

Function: `double reseptivity(int which)`

Input: `which-->` The spin you wish to extract

Description: `Returns the relative sensitivity of the nucleous RELATIVE TO 13C.`

Example `SpinSys moo(6);`

Usage: `cout<<moo.reseptivity(2); //prints out '5.68E+3' (i.e. it has a signal 5.68E+3 times stronger then 13C)`

SpinSys::element extraction

Function:

□ **std::string symbol(int which)**

Input:

□ which--> The spin you wish to extract

Description:

□ Returns the symbol name of the spin 'which.'

Example Usage:

□ `SpinSys moo(4);
cout<<moo.symbol(2); //prints out '1H'`

SpinSys::element extraction

Function: `double weight(int which)`

Input: `which-->` The spin you wish to extract

Description: `Returns the mass in atomic Mass Units (amu) of the NATURAL abundance.`

Example Usage: `SpinSys moo(6);
cout<<moo.weight(1); //prints out '1.008665'`

SpinSys::element extraction

Function: **double weight()**

Input: void

Description: Returns the mass in atomic Mass Units (amu) of the NATURAL abundance of TOTAL SYSTEM.

Example SpinSys moo(6);

Usage: cout<<moo.weight(); //prints out '6.01'

SpinSys::assignments

Function: `SpinSys &operator=(const SpinSys &rhs);`

Input: `rhs-->` the item you wish to copy...

Description: Assigns one SpinSys from another. This also copies all the Spinoperators already calculated.

Example Usage: `SpinSys moo(2);`

`SpinSys koo;`

`koo=moo;`

SpinSys::assignments

Function: `SpinSys &operator=(const spin_sysPars &sym);`

Input: `sym-->`another set of 'parameters' of the spins system (the spin_sysPars simply act a list mechanism for the Isotope)

Description: `Assigns the SpinSys from the symbol name.`

Example `spin_sysPars joo(4);`

Usage: `SpinSys moo=joo;`

SpinSys::assignments

Function:

- `Isotope &operator()(int which);`
- `Isotope &operator[](int which);`
- `Isotope operator()(int which) const;`
- `Isotope operator[](int which) const;`

Input:

- `which--> the isotope at position 'which'`

Description:

- This can be used to access any Isotope related functions AND also for assignment of isotope types inside the SpinSys.

Example

- `SpinSys moo(2);`

Usage:

```
moos(0)="13C"; //set the first spin to be a 13C
moos[1]="1H"; //set the second spin to be a 1H
double 13cg=moo(0).gamma(); //the gamma factor for the first
spin...
```

SpinSys::Math

Function:

- `bool operator==(const SpinSys &rhs)`
- `bool operator!=(const SpinSys &rhs)`

Input:

- `rhs-->` another SpinSys

Description:

- Determines weather or not the rhs is the same SpinSys as the lhs.

Example Usage:

```
□ SpinSys moo(2), loo(2);
moo!=loo; //would be false
moo(1)="13C";
moo!=loo; //true
moo==loo; //false
```

SpinSys::IO

Function: `o ostream &operator<<(std::ostream &oo, SpinSys &iso)`

Input: `o Iso--> the SpinSys you wish to output`
`oo--> an output stream`

Description: `o --writes the string....`
`Spins: 1 2 3 ...`
`Isotope: 1H 1H 1H....`
`Momentum: 1/2 1/2 1/2`

Example Usage: `o SpinSys mo(2);`
`mo(1) = "13C" ;`
`cout << mo ;`

SpinSys::other

Function: **rdmatrix &F0()**
 rimatrix &Fe()
 rmatrix &Fmi()
 rmatrix &Fp()
 rdmatrix &Fz()
 smatrix &Fx()
 hmatrix &Fy()

Input: void

Description: TOTAL space spin operators. These are generated the first time ANY F{i} is called and only generated ONCE.

Example SpinSys moo(2);

Usage: moo.Fx(); // Fx=Ix(0)+Ix(1)

SpinSys::other

Function:

- **bool heteronuclear()**

Input:

- void

Description:

- TRUE-->if the system is heteroculear
FALSE--> NOT heteroculear

Example Usage:

- ```
SpinSys moo(2);
moo.heteronuclear(); //false
moo(1)="13C";
moo.heteronuclear(); //true
```

## SpinSys::other

### Function:

**bool homonuclear()**

### Input:

none

### Description:

TRUE-->if the system is homonuclear  
     FALSE--> NOT homoculear

### Example Usage:

SpinSys moo(2);  
        moo.homonuclear(); //true  
        moo(1)="<sup>13</sup>C";  
        moo.homonuclear(); //false

## SpinSys::other

### Function:

- **rimatrix Ie(int which)**
- **rmatrix Imi(int which)**
- **rmatrix Ip(int which)**
- **rdmatrix Iz(int which)**
- **smatrix Ix(int which)**
- **hmatrix Iy(int which)**

### Input:

- which--> the desired spin matrix..

### Description:

- Individual spin operators in the Hilbert space of the TOTAL system.

### Example Usage:

- `SpinSys moo( 2 );`
- `moo.Ix( 0 );`

## SpinSys::other

**Function:**  **int isotopes()**

**Input:**  none

**Description:**  Returns the number of DIFFERENT isotopes in th system.

**Example Usage:**  SpinSys moo(2);  
cout<<moo.isotopes(); //prints out '1'  
moo(1)="13C";  
cout<<moo.isotopes(); //print out '2'

## SpinSys::other

|                  |                                                                                                                                                                                                                                 |
|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b> | <code>bool isotopes(std::string in)</code>                                                                                                                                                                                      |
| Input:           | <code>in--&gt; a string that would be a valid isotope label (i.e. 1H, 13C, etc)</code>                                                                                                                                          |
| Description:     | <ul style="list-style-type: none"><li>□ Returns true IF 'in' is an isotope in the system and FALSE if not.</li></ul>                                                                                                            |
| Example Usage:   | <ul style="list-style-type: none"><li>□ <code>SpinSys moo(2);</code></li><li>    <code>moo.isotopes("1H"); //true</code></li><li>    <code>moo(1)="13C";</code></li><li>    <code>moo.isotopes("14N"); //False</code></li></ul> |

## A Simple Spin System

---

The SpinSys is the link between the Isotope and spin operators. All functions in Isotope are available here, except with indices required. The Spin operators are also generated by this class and it tries to minimize their calculation for speeds sake.

---

```
//sets up a data structure with all the needed info for 3 spins...
SpinSys mySys(3);

//set the second spin to a 13C
mySys(1)="13C";

//prints the gamma factor for the second spin
cout<<mySys.gamma(1);
//obtains the TOTAL Ix operator
//(the operator is generates ONLT once...)
rmatrix fx=mySys.Fx(); //a new spin type with a new Hilbert Space...
//thus i need to call GenSpinOps again
mySys(0)="27Al";
//this 'fx' HAS NOT BEEN UPDATED!!!!!!...SET YOUR SPINS
//TYPES BEFORE CALLING OR call GenSpinOps()
//After you change the spins....
fx=mySys.Fx(); mySys.GenSpinOps();

//NOW this is the correct Total spin operator
fx=mySys.Fx();
/*
I can Also avoid calling GenSpinOps by using the functions
in the SpinOperator section...this should be done if 1)
you use the spinoperators only a few times or 2)
you only need to use a few spin operators or 3)
your spin system is really huge and you have little memmory
*/
fx=Ix(mySys, 2); //the Ix operator for the second spin...
```

---

## --Global Functions Global Spin operator Functions

---

--- Ie

--- Imi

--- Ip

--- Ix

--- Iy

--- Iz

Examples

--- 1) simple usage

These are simple a series of funcitons that generate matrix representations of the standard spin operators (Iz, Ix, Iy, Ip, Imi, Ie) for both a 'Hilbert Space size' (i.e. spin 1/2, or spin 1, etc) AND composite spin spaces (i.e. the Ix operator for a 13C in the space of 13C-14N together...).

- These functions generate these operator ONCE upon calling, and save them in a 'map' so that if used again, they do not have to be regenerated.
-

## SpinOperators::global function

### Function:

□ **rimatrix Ie(int Hs);**  
    **rimatrix Ie(SpinSys A, int spin);**

### Input:

□ Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

### Description:

□ Returns Identity matrix of size HsxHs (a real identity matrix).

### Example Usage:

□ cout<<Ie(3); //prints out '(1 0 0), (0 1 0), (0 0 1)'

## SpinOperators::global function

**Function:**  **rmatrix Imi(int Hs)**  
 **rmatrix Imi(SpinSys &A, int spin)**

**Input:**  Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

**Description:**  Returns Imi matrix of size HsxHs (a real diagonal matrix). If given a SpinSys it returns Ix matrix of the TOTAL spin system size.

**Example**  cout<<Imi(2); //prints out '(0 0), (1 0)'

**Usage:** SpinSys A(3);  
cout<<Imi(A,2); //prints out the Im matrix (6x6) for the third spin

## SpinOperators::global function

**Function:**  **rmatrix** Ip((int Hs);  
**rmatrix** Ip(SpinSys A, int spin)

**Input:**  Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

**Description:**

**Example**  cout<<Ip(2); //prints out '(0 1), (0 0)'

**Usage:** SpinSys A(3);

cout<<Ip(A,2); //prints out the Ip matrix (6x6) for the third spin

## SpinOperators::global function

**Function:**  **smatrix Ix(int Hs);**  
 **smatrix Ix(SpinSys A, int spin)**

**Input:**  Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

**Description:**  Returns Ix matrix of size HsxHs (a real symmetric matrix). If given a SpinSys it returns Ix matrix of the TOTAL spin system size.

**Example**  SpinSys A(3);

**Usage:**  cout<<Ix(A,2); //prints out the Ix matrix (6x6) for the third spin  
cout<<Ix(2); //prints out '(0 1), (1 0)'

## SpinOperators::global function

**Function:**  **hmatrix Iy(int Hs);**  
**hmatrix Iy(SpinSys A, int spin)**

**Input:**  Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

**Description:**  Returns Iy matrix of size HsxHs (a hermitian matrix). If given a SpinSys it returns Ix matrix of the TOTAL spin system size.

**Example**  cout<<Iy(2); //prints out '(0 complexi), (-complexi 0)

**Usage:**

```
SpinSys A(3);
cout<<Iy(A,2); //prints out the Iy matrix (6x6) for the third spin
```

## SpinOperators::global function

**Function:**  **rdmatrix Iz((int Hs);**  
**rdmatrix Iz(SpinSys A, int spin)**

**Input:**  Hs--> Hilbert Space size  
A--> A 'spin\_system'  
spin--> the spin NUMBER you wish to obtain the operator for

**Description:**  Returns Iz matrix of size HsxHs (a real diagonal matrix). If given a SpinSys it returns Ix matrix of the TOTAL spin system size.

**Example**  cout<<Iz(2); //prints out '(1 0), (0 -1)

**Usage:**

```
SpinSys A(3);
cout<<Iz(A, 2); //prints out the Iz matrix (6x6) for the third spin
```

## A simple useage demonstration

---

Very basic usage of the gloabl functions.

---

```
rmatrix mIx=Ix(3); //generates a 3x3 Ix matrix
rdmatrix mIe=Ie(5); //generates a 5x5 Identity matrix

SpinSys A(5); //a 5 spin spin system
A(2)="13C"; //set the third spin to a Carbon 13

//The I+ operator for the third spin
//in 'A' in the hilbert space of all three spins...
matrix mIp2=Ip(A,2);

//The I- operator for the third spin in 'A'
//in the hilbert space of all three spins...
matrix mIm2=Imi(A,2);
```

---

--Global Functions Various Spin and Spacial Tenors to second rank

--- A1

--- A2

--- A\_CSA\_pas

--- A\_Dip\_pas

--- A\_Q\_pas

--- T0

--- T1

--- T11, T10, T1m1

--- T2

--- T22, T21, T20,

T2m1, T2m2

--- T2\_rot

--- T\_CSA

--- T\_Dip

--- T\_J

--- T\_Quad

--- wigner1

--- wigner2

### Examples

--- space tensors

--- Spin tensors

---

These are simple a series of funcitons that generate matrix representations of the SPIN SPHERICAL TENSORS up to second rank...these REQUIRE a 'SpinSys' to function properly

The Spacial tensors are simply 'numbers' and only require angles.

---

## Spin/Space Tensors::global function

**Function:** `double A1(double alpha, double beta, int m)`

**Input:**

- [alpha, beta, gamma] --> A euler angle rotation in RADIANS
- m--> the z projection (m= -1, 0,1)

**Description:**

- The first rank m'=0 wigner rotations...this is a short hand for 'wigner1(a, b, 0, 0, m)'

**Example Usage:** `double a=78*Pi/180, b=45*PI/180, g=67*Pi/180;  
cout<<A1(a,b,1);`

## Spin/Space Tensors::global function

**Function:** `double A2(double alpha, double beta, int m)`

**Input:**

- [alpha, beta, gamma] --> A euler angle rotation in RADIANS
- m--> the z projection (m= -2,-1, 0,1,2)

**Description:**

- The second rank m'=0 wigner rotations...this is a short hand for 'wigner2(a, b, 0, 0, m)'

**Example Usage:** `double a=78*Pi/180, b=45*PI/180, g=67*Pi/180;  
cout<<A2(a,b,2);`

## Spin/Space Tensors::global function

**Function:** `double A_CSA_pas(double iso, double del, double eta,int l, int m)`

**Input:**

- iso--> isotropic shift (Hz)
- del-->anisotropy (Hz)
- eta--> the assymmetry parameter for a csa
- l--> the RANK desired
- m--> the z projection (m=-2,-1,0,1,2)

**Description:**

- Returns the proper coef to be placed with 'T\_CSA' to generate a hamiltonian.

```
A_CSA_pas(iso, del, eta, 0,0)=-sqrt(3)*iso;
A_CSA_pas(iso, del, eta, 1, 1..0..-1.)=0
A_CSA_pas(iso, del, eta, 2,2)=1./2.*del*eta;
A_CSA_pas(iso, del, eta, 2,-2)=1./2.*del*eta;
A_CSA_pas(iso, del, eta, 2,1)=0
A_CSA_pas(iso, del, eta, 2,0)=del
```

**Example**

- `cout<<A_CSA_pas( 3000, 2000, 0, 2, 2);`

**Usage:**

## Spin/Space Tensors::global function

### Function:

□ **double A\_Dip\_pas(double del, int m)**

### Input:

□ del-->anisotropy (Hz) (the dipolar coupling)  
m--> the z projection (m=-2,-1,0,1,2)

### Description:

□ Returns the proper coef to be placed with 'T\_Dip' to generate a hamiltonian.

A\_Dip\_pas( del, (-2,-1,1,2))=0;  
A\_Dip\_pas( del,0)= -sqrt(6)\*dip;

### Example Usage:

□ cout<<A\_Dip\_pas( 3000 , 0 );

## Spin/Space Tensors::global function

**Function:** `double A_Q_pas(double eta, int m)`

**Input:**

- eta--> the assymmetry parameter for a quadrupole
- m--> the z projection (m=-2,-1,0,1,2)

**Description:**

- Returns the proper coeif to be placed with 'T\_Q' to generate a hamiltonian to only the 'first order.'

**Example**

- `cout<<A_Q_pas(0.1, 2);`

**Usage:**

## Spin/Space Tensors::global function

**Function:**  **rmatrix T0(SpinSys A, int on)**

**Input:**  A-->A valid spinsystem  
on--> the spin you wish to grab the Zeroth rank spin tensor

**Description:**  Returns Identity matrix of size HsxHs (a real identity matrix) (the T0 is the indentitiy matrix).

**Example Usage:**  SpinSys A(3);  
cout<<T0(A,1);

## Spin/Space Tensors::global function

### Function:

□ **rmatrix T1(SpinSys A, int on, int m)**

### Input:

- A-->A valid spinsystem
- on--> the spin you wish to grab the first rank spin tensor
- m--> the projection along the rank axis..(m=-1, 0, or 1)

### Description:

- Returns a matrix (full complex) from the spin system given 'm.'
- m=1--> T1(A, on, 1)=Ip(a, on);
- m=0--> T(A, on, 0)=Iz(A, on);
- m=-1--> T(A, on, -1)=-Imi(A, on);

### Example Usage:

□ `SpinSys A( 3 );  
cout<<T1(A, 2, 0);`

## Spin/Space Tensors::global function

### Function:

- **rmatrix T11(SpinSys A, int on);**
- **rdmatrix T10(SpinSys A, int on);**
- **rmatrix T1m1(SpinSys A, int on);**

### Input:

- A-->A valid spinsystem
- on--> the spin you wish to grab the first rank spin tensor

### Description:

- Returns a matrix (full complex) from the spin system given 'm.'
- T11--> T1(A, on, 1)=Ip(a, on);
- T10--> T(A, on, 0)=Iz(A, on);
- T1m1--> T(A, on, -1)=-Imi(A, on);

### Example Usage:

- ```
SpinSys A(3);
cout<<T10(A,2);
```

Spin/Space Tensors::global function

Function: **matrix T2(SpinSys A, int on,int m);** -->Quadrupolar type tensors
matrix T2(SpinSys A, int on1,int on2,int m); -->Dipolar type tensors

Input: A-->A valid spinsystem
on--> The makes the assumption that on1=on2 (this will return a zero matrix IF the spin's momentum is NOT>1/2 (i.e. a quadropole)

on1--> the FIRST spin you wish to grab the second rank spin tensor
on2--> the SECOND spin you wish to grab the second rank tensor

m--> the projection along the rank axis..(m=-2,-1, 0, 1, or 2)

Description: These are 2 spin/second rank tensors and require two indexes....
--if the spin qn>1/2 then on1==on2 will give the second rank quadrupole tensors
--Returns a matrix (full complex) from the spin system given 'm'

m=2 --> T2(A, on1, on2, 2)= 1/2(A.Ip(on1)*A.Ip(on2));
m=1--> T2(A, on1, on2, 1)=-1./2.* (A.Ip(on1)*A.Iz(on2)+A.Iz(on1)*A.Ip(on2));
m=0--> T2(A, on1, on2, 0)=1./sqrt(6.)*(2*A.Iz(on1)*A.Iz(on2)-
(A.Ix(on1)*A.Ix(on2)+A.Iy(on1)*A.Iy(on2)));
m=-1--> T2(A, on1, on2, -1)=1./2.* (A.Imi(on1)*A.Iz(on2)+A.Iz(on1)*A.Imi(on2));
m=-2-->T2(A, on1, on2, -2)=1/2(A.Imi(on1)*A.Imi(on2));

Example SpinSys A(3);
Usage: cout<<T2(A , 1 , 2 , 0) ;

Spin/Space Tensors::global function

Function:

- **rmatrix** T22(SpinSys A, int on);--> Quadrupolar type tensors
- rmatrix T22(SpinSys A, int on1, int on2); --> Dipolar type tensors
- rmatrix T21(SpinSys A, int on); -->Quadrupolar type tensors
- rmatrix T21(SpinSys A, int on1, int on2); --> Dipolar type tensors
- matrix T20(SpinSys A, int on); -->Quadrupolar type tensors
- matrix T20(SpinSys A, int on1, int on2); --> Dipolar type tensors

- rmatrix** T2m1(SpinSys A, int on) ;-->Quadrupolar type tensors
- rmatrix** T2m1(SpinSys A, int on1, int on2); -->Dipolar type tensors

- rmatrix** T2m2(SpinSys A, int on); -->Quadrupolar type tensors
- rmatrix** T2m2(SpinSys A, int on1, int on2); --> Dipolar type tensors

Input:

- A-->A valid spinsystem
- on--> the spin you wish to grab the Second rank spin tensor
- on1--> the FIRST spin you wish to grab the second rank spin tensor
- on2--> the SECOND spin you wish to grab the second rank tensor

Description:

- Returns the Second rank spin tensors

```
T22= 1/2(A.Ip(on1)*A.Ip(on2));
T21=-1./2.*(A.Ip(on1)*A.Iz(on2)+A.Iz(on1)*A.Ip(on2));
T20=1./sqrt(6.)*(2*A.Iz(on1)*A.Iz(on2)-(A.Ix(on1)*A.Ix(on2)+A.Iy(on1)*A.Iy(on2)));
T2m1=1./2.*(A.Imi(on1)*A.Iz(on2)+A.Iz(on1)*A.Imi(on2));
T2m2=1/2(A.Imi(on1)*A.Imi(on2));
```

Example

- SpinSys A(3);
- cout<<T22(A,2,1);
- A(2)="14N";
- cout<<T22(A,2);

Usage:

Spin/Space Tensors::global function

Function: **matrix T2_rot(SpinSys A,int on, int m, double alpha, double beta, double gamma);** --> Quadrupolar type tensors

matrix T2_rot(SpinSys A,int on1, int on2, int m, double alpha, double beta, double gamma); --> Dipolar type tensors

Input: A-->A valid spinsystem

on--> the spin you wish to grab the Second rank spin tensor

on1--> the FIRST spin you wish to grab the second rank spin tensor

on2--> the SECOND spin you wish to grab the second rank tensor

[alpha, beta, gamma] --> a EULER angle rotation Rz(alpha)Ry(beta)Rz(gamma) in RADIANS

Description: Returns the rotated second rank spin tensor

T2_rot=Sum_m' [Am'm(alpha, beta, gamma)*T2m']

T2_rot= T22*A2m(a,b,g)+T21*A1m(a,b,g)+T20*A0m(a,b,g)+T2-1*A-1m(a,b,g)+T2-2*A-2m(a,b,g)

Where the Am'm are the spacial tensors

Example SpinSys A(3);

Usage: double a=45*PI/180, b=65*PI/180, c=87*PI/180

cout<<T2_rot(A,0,1,2, a,b,c); //returns the ROTATED T22 spintensor
A(2)="14N";

cout<<T2_rot(A,2,-1, a,b,c); //returns the ROTATED T2m1 quadrupole tensor

Spin/Space Tensors::global function

Function: `rmatrix T_CSA(SpinSys A, int spin, int l, int m);`

Input: `A--> SpinSystem`

`spin--> the spin NUMBER you wish to obtain the operator for`

`l--> the desired RANK (l=0,1,2)`

`m-->the desired z projection (for l=0, m=0, for l=1, m=(-1,0,1), for l=2, m=(-2,-1,0,1,2)`

Description: `>Returns The Spherical tensors for a CSA (Chemical Shift Anisotropy) IN A HIGH MAGNETIC FIELD ALONG THE Z AXIS!! (a typical NMR experiment...we still have T2 components as the 'second spin' is considered to be the magnetic field)`

`T_CSA(A, spin, 0,0)=1/sqrt(3)*A.Iz(spin); //T00`

`T_CSA(A, spin, 1,(-1,0,1))=0; //T1m1, T10, T11`

`T_CSA(A, spin, 2,2)=0; //T22`

`T_CSA(A, spin, 2,-2)=0; //T2m2`

`T_CSA(A, spin, 2,1)=-1./2.* (A.Ip(spin)); //T21`

`T_CSA(A, spin, 2,-1)=-1./2.* (A.Imi(spin)); //T2m1`

`T_CSA(A, spin, 2,0)=(A.Iz(spin)); //T20`

Example `SpinSys A(3);`

Usage: `cout<<T_CSA(A, 2 , 2 ,-1); //prints out the T2m1 of the CSA for spin 2`

Spin/Space Tensors::global function

Function: `hmatrix T_Dip(SpinSys A, int spin1, int spin2, int m);`

Input: `A--> SpinSystem`

`spin1--> the spin NUMBER of the FIRST spin you wish to obtain the operator for`

`spin2--> the spin NUMBER of the SECOND spin you wish to obtain the operator for`

`m-->the desired z projection (m=(-2,-1,0,1,2))`

Description: `>Returns The Spherical tensors for a Dipole-Dipole Interaction IN A HIGH MAGENTIC FIELD ALONG THE Z AXIS!! (a typical NMR experiment...)`

There are NO T00, or T11 compents for a dipole.

IF Spin1 == Spin2 then we get the HOMONUCLEAR version of the dipole tensors

IF Spin1 != Spin2 then we get the HETERONUCLEAR veriosn of the dipole tensors

SpinSys A(3);

```
T_Dip(A, spin1,spin2, 2)=0; //T22
T_Dip(A, spin1,spin2, -2)=0; //T2m2
T_Dip(A, spin1,spin2,1)=0; //T21
T_Dip(A, spin1,spin2,-1)=0//T2m1
T_Dip(A, spin1,spin2,0)=sqrt(1/6)*(2.*A.Iz(spin1)*A.Iz(spin2)-A.Ix(spin1)*A.Ix(spin2)-
A.Iy(spin1)*A.Iy(spin2));
//T20 --> HOMONUCLEAR
=sqrt(2/3)*(A.Iz(spin1)*A.Iz(spin2)); //T20 --> HETERONUCLEAR
```

Example

`SpinSys A(3);`

Usage:

```
cout<<T_Dip(A,1, 2,0); //prints out the T20 of the Dipole for spin
2 and spin 1
```

Spin/Space Tensors::global function

Function: `hmatrix T_J(SpinSys A, int spin1,int spin2, int m=0, int strong=0);`

Input: `A--> SpinSystem`

`spin1--> the spin NUMBER of the FIRST spin you wish to obtain the operator for`

`spin2--> the spin NUMBER of the SECOND spin you wish to obtain the operator for`

`m-->the desired z projection (only m=0 give a non zero answer)`

`strong-->string J coupling==1, weak J coupling==0`

Description: `>Returns The Spherical tensors for a J coupling IN A HIGH MAGENTIC FIELD ALON THE Z AXIS!! (a typical NMR experiment...`

For the scalar coupling we only have T00...but there are two extreams

1) if the chemical shift between th two spins is much bigger then the J coupling we have the 'weak' coupling limit where the J interaction is truncated with respect to The Chemical shift resulting in an ' Iz_1*Iz_2 ' spin operator

2) if the chemical shift is small or comparable to J we must leave The J interaction as it originally stands.. $I_1 * I_2 (Iz*Iz+Iy*Iy+Ix*Ix)$

`SpinSys A(3);`

`//weak coupling...`

`T_J(A, spin1, spin2,0)=1/sqrt(3)*A.Iz(spin1)*A.Iz(spin2); //T00`

`//strong coupling...`

`T_J(A, spin1, spin2,0)`

`=1/sqrt(3)*(A.Iz(spin1)*A.Iz(spin2)+A.Iy(spin1)*A.Iy(spin2)+A.Ix(spin1)*A.Ix(spin2)); //T00`

Example `SpinSys A(3);`

Usage: `cout<<T_J(A, 1, 2); //prints out the T00 of the J for spin 1 and 2-->Iz(1)*Iz(2)/sqrt(3)`

Spin/Space Tensors::global function

Function: `rmatrix T_Qquad(SpinSys A, int spin, int m, int Order=1);`

Input:

- `A--> A SpinSystem`
- `spin1--> the spin NUMBER of the FRIST spin you wish to obtain the operator for`
- `Order--> the ORDER of the tensors...`
- `m-->the desired z projection (m=(-2,-1,0,1,2))`

Description: `Order=1--> the 'first order' approximation of the rotating wave/high field perturbation...`
`(only m=0 gives a non-0 compenent (i.e. T20)`

`Order=2--> the the terms from the second order perterbation theory`

`m=0 gives the First Order component...`

`m=11 give the 'rank 1' SECULAR commutators--> [T2m1,T21]=[Iz*Imi+ Imi*Iz,Iz*Ip+Ip*Iz])`

`m=22 gives the 'rank 2' SECULAR commutators-->[T2m2, T22]=[Imi*Imi,Ip*Ip]`

Returns The Spherical tensors for a Quadrupole Interaction IN A HIGH MAGENTIC FIELD
ALONG THE Z AXIS!! (a typical NMR experiment...)

NOTE:: there are NO T00, or T1m compents for a quadrupole

SpinSys A(3);

`//order=1`

`T_Qquad(A, spin, 2)=0; //T22`

`T_Qquad(A, spin, -2)=0; //T2m2`

`T_Qquad(A, spin,1)=0; //T21`

`T_Qquad(A, spin,-1)=0//T2m1`

`T_Qquad(A, spin,0)=sqrt(1/6)*(3.*A.Iz(spin)*A.Iz(spin)-A.qn(spin)*(A.qn(spin)-1)*A.Ie(spin));`

`//order=2`

`T_Qquad(A, spin, 2,2)=0; //T22`

`T_Qquad(A, spin, -2,2)=0; //T2m2`

`T_Qquad(A, spin,1,2)=0; //T21`

`T_Qquad(A, spin,-1,2)=0//T2m1`

`T_Qquad(A, spin,0,2)=3.*A.Iz(spin)*A.Iz(spin);`

`T_Qquad(A, spin,11,2)=T2m1*T21-T21*T2m1`

`T_Qquad(A, spin,22,2)=T2m2*T22-T22*T2m2`

Example

`Spinsys A(3);`

Usage:

`A(2)="14N";`

`cout<<T_Qquad(A,1, 2,0); //prints out the T20 of the Quad for spin 2`

Spin/Space Tensors::global function

Function: **complex wignerl(double alpha, double beta, double gamma,int m, int mp)**

Input: [alpha, beta, gamma] --> A euler angle rotation in RADIANS
m--> the matrix element row(m= -1, 0,1)
mp--> the matrix element colum (mp=-1,0,1)

Description: The first rank wigner matrix elements... (too long to write here...look to 'src/QMspins/space_ten.cc' for the values.

Example double a=78*Pi/180, b=45*PI/180, g=67*Pi/180;

Usage: cout<<wignerl(a,b,g,1,0);

Spin/Space Tensors::global function

Function: **complex wigner2(double alpha, double beta, double gamma,int m, int mp)**

Input: [alpha, beta, gamma] --> A euler angle rotation in RADIANS
m--> the matrix element row(m= -2,-1, 0,1,2)
mp--> the matrix element colum (mp=-2,-1,0,1,2)The

Description: Second rank wigner matrix elements... (too many to write here...look to 'src/QMspins/space_ten.cc' for the values.

Example double a=78*Pi/180, b=45*PI/180, g=67*Pi/180;
Usage: cout<<wigner2(a,b,g,2,1);

Demonstration of using the Spacial rotation tensors and the Rotation class.

```
//A10=sqrt(3/4)*cos(theta)--> first rank m=0, m'=0
double a1=A1(34*Pi/180, 45*PI/180, 0);
//A20=1/2*(3cos(theta)^2-1) --> second rank m=0, m'=0
double a2=A2(34*Pi/180, 45*PI/180, 0);

double alpha=89*Pi/180, beta=23*PI/180, gamma=89/180;
//The first rank wigner rotation matrix element '1,0'
double wl=wigner1(alpha, beta, gamma, 1,0);
//The second rank wigner rotation matrix element '1,-2'
double wl=wigner2(alpha, beta, gamma, 1,-2);

//The rotation class is used throughout the 'specific' interaction (Dip, Quad, J, CSA)
//TO generate the hamiltonians fast and efficiently...
// things like the powder angles and spinner angle rotations are the same
// for each interaction...this class acts as the container to pass these values around
Rotation Rots; //set up a 'rotation class'
Rots.setPowderAngles(alpha, beta, gamma); //set up the powder angle rotations matrix

double wr=5000, t=1e-3, magang=54.7*PI/180.
//set the spinner rotation vector
Rots.setRotorAngles(wr*t*PI2, magang);

//get one of the rotation bits...
double pA12=Rots.powderAs(1,2);
//get the A20 part of the spinning Rotation elements
double sA20=Rots.spinnerAs(2);
```

Using the Spin Tensors

This is a simple demonstration on how to use the spin tensors.

```
SpinSys A(5); //a 5 spin spin system
//set the third spin to a Carbon 13
A(2)="13C";
//set the third spin to a Nitrogen 14 (a quadrupole)
A(3)="14N";
//The T22=(Ip1*Ip2)/sqrt(2) second rank tensor
//between spin 1 and spin 2 of system A
matrix m2=T2(A,1,2,2);
//The T22=(Ip1*Ip2)/sqrt(2) second rank
// tensor between spin 1 and spin 2 of system A
m2=T22(A,1,2);
//The T22=(Ip1*Ip2)/sqrt(2) second rank tensor
//For the Quadrupole of system A
m2=T22(A,3);

//The T22m=(Im1*Im2)/sqrt(2) second rank
//tensor between spin 1 and spin 2 of system A
matrix m2m=T2(A,1,2,-2);
//The T22m=(Im1*Im2)/sqrt(2) second rank
//tensor between spin 1 and spin 2 of system A
m2m=T2m2(A,1,2);
//The T22m=(Im1*Im2)/sqrt(2) second rank
// tensor For the Quadrupole of system A
m2m=T2m2(A,3);
//The T11=-Imi second rank tensor
//between spin 4 of system A
matrix m1=T1(A,4,1);
//The T11=-Imi second rank tensor between
//spin 4 of system A
m1=T11(A,4);
//The T10=Iz1 second rank tensor between
// spin 1 of system A
matrix m0=T1(A,1,0);
//The T10=Iz1 second rank tensor between
//spin 1 of system A
m0=T10(A,1);
```

--Global Functions class Rotations { ...

--- rotationMatrix3D --

--Public Vars

--- alpha, beta, chi

--- cartPower

--- cartSpin

--- phi, theta, gamma

--- powderAs

--- RotationType

--- spinnerAs

--Constructors

--- Rotations

--Other Functions

--- setPowderAngles

--- setRotorAngles

Examples

--- Zero Field Dipole

- The idea of the class is to simplify the syntax for multiple tensorial rotation and/or cartesian of spin interactions.
In a High magnetic field we only care about the FINAL m=0 component of the resulting rotation so
- Using Spherical Tensors.
 - For any give Spin interaction in the Solid State we can have up to 3 different rotations going from the crystal frame to the lab frame
 1. PAS-->Molecule (rotation from principle axis frame to the molecule frame)
 - These rotation are handled by the 'Interaction Class' (Dip, Quad, J, CSA) and take the static values of 'eta', 'del', 'iso' and rotate them into a '5 vector' i.e. (A-2, A-1, A0, A1, A2) of the MOLECULE FRAME which will be called "crystalAs".
 - If there is no relative orientation then this five vector is typically the coupling value (see the A_Dip_pas, etc as an example)
 2. Molecule-->Rotor (rotation of the molecule into a powder frame with respect to the rotor)
 - If the sample is not physically spinning then we only need the Final 'A20' term after the rotation and can be represented by

A20=Sum_m[D_0,m * crystalAs[m]]

Where 'D' is the wigner rotation matrix

NOTE:: This 'D' is the same for all Spins and interactions for a given powder angle so we need only to calculate it once...the Rotations Class calculates it via...

Rotation::setPowderAngles(double phi, double theta,
double gamma)
and is stored in the class matrix 'powderAs'

3. So our final A20 term is calculated via

```
A20=Rot.powderAs(0,2)*crystalAs[0] //wigner2(-2,0)*A(-2)
+Rot.powderAs(1,2)*crystalAs[1] //wigner2(-1,0)*A(-1)
+Rot.powderAs(2,2)*crystalAs[2] //wigner2(0,0)*A(0)
+Rot.powderAs(3,2)*crystalAs[3] //wigner2(1,0)*A(1)
+Rot.powderAs(4,2)*crystalAs[4]; //wigner2(2,0)*A(-2)
```

This term is then used to multiply the T20 spin tensor... and our hamiltonian is calculated

H= A20*T20 (in a High field)

- Rotor-->Lab (if the rotor is spinning then each crystallite is also spinning)
 - Here the calculation gets more involved, the Powder wigners that we could ignore in the last rotation now get mixed into a NEW A20...but as we are still only interested in the final m=0 part we can write our total rotation as

A20=Sum_mp[D_0,mp(wr, beta)*Sum_m [D_mp,m(a,b,g) * crystalAs(m)]]

To calculate the nessesary Spinning Terms we use

```
//set the spinner wigners
Rotation::setSpinnerAngles(double alpha, double theta)

alpha is typically the rotor speed * time (*2Pi)
beta is the axis the rotor makes with the B_o field.
```

The above equation can written in the code for is

```
A20=Sum_mp [ Sum_m
[Rot.spinnerAs(mp)*Rot.powerAs(mp,m)*crystalAs(m)) ] ]
```

again out Hamiltonian is This A20*T20...

- These 'spinnerAs' are also the same for every spin and interaction (they only depend on the rotor) so this class provides a nice way to share in the info AND explicitly show people the rotations you are performing....

- Using Cartesian Rotations

- As in the Spherical case, we still have 3 different rotations, however, the rotations are implemented differently.

1. PAS-->Molecule (rotation from principle axis frame to the molecule frame)

- The PAS of the system can now be represented as a 3x3 tensor (rather than the 5 vector of spherical tensors)

For instance a Quadrupole has a Cartesian PAS frame of

```
[ (eta-1)/2, 0, 0]
[ 0, -(eta+1)/2, 0]
[ 0, 0, 1]
```

- To rotate this initial tensor into the Molecule frame, we simply need a 3D rotation matrix, called via

```
rmatrix Rot=rotationMatrix3D(alpha, beta, gamma);
```

This provides a 3x3 rotation matrix, and the new molecule frame is just

```
rmatrix Molecule=Rot*PAS*transpose(Rot);
```

2. These operations are handled INSIDE the Spin-Interaction (Csa, Quad, Dip, etc) classes, NOT the Rotations class, as they only need to typically be applied once during an simulation.

- Molecule-->Rotor (rotation of the molecule into a powder frame with respect to the rotor)

- This is where the Rotations Class is used.
- All we need to do is to create a Rotation Matrix from the Molecule to the Rotor via

```
Rotations::setPowderAngles(alpha, beta, gamma).
```

This assumes that you have set the RotationsType=Rotations::Cartesian, or
Rotations::All.

- Then we rotate the Molecule via

```
rmatrix
Rotor=(Rotations::cartPowder)*Molecule*transpose(Rotations::cartPowder);
```

- Should we not need any more Rotations (i.e. not spinning) then to calculate the hamiltonian, all we need is the final state of the Z element of the resulting matrix

```
H=Rotor(2,2)*{interaction strength}*T20
```

- Rotor-->Lab (if the rotor is spinning then each crystallite is also spinning)

- Again, we simply need the 3D rotation matrix from the Rotor to the lab via

```
Rotations::setRotorAngles(alpha, beta, gamma=0).
```

This assumes that you have set the RotationsType=Rotations::Cartesian, or
Rotations::All.

- Then we rotate the Molecule via

```
rmatrix
Final=(Rotations::cartSpin)*Rotor*transpose(Rotations::cartSpin);
```

- Should we not need any more Rotations (i.e. not spinning) then to calculate the hamiltonian, all we need is the final state of the Z element of the resulting matrix

```
H=Final(2,2)*{interaction strength}*T20
```

- For the second order Quadrupole, other matrix elements from Final are used.
- Some Numerical Notes::

- Using the cartesian space rotations is more efficient than the Spherical rotations in general (i.e. if you are spinning and have powder angles). But for special cases and interactions (i.e. a non spinning dipole-dipole interaction) the Hamiltonian is faster

generated using the spherical rotations. The Interaction classes (Quad, Dip, Csa) have been tuned for such optimal performances

!!NOTE!!

- Things to note::
 - The treatment for second order quadrupoles is different than what is shown here because we have a mixture of each sub rotation element for the nasty details look to 'src/QMspins/qua.cc'
-

Rotations::global function

Function: **rmatrix rotationMatrix3D(double a, double b, double g);**

Input: a--> the alpha Euler angle in RADIANS
 b--> the beta Euler angle in RADIANS
 g--> the gamma Euler angle in RADIANS

Description: Return a 3x3 real matrix containing the elements to rotate a 3-vector (or another 3x3 tensor) by the Euler angles [a,b,g].

Example rmatrix myRot=rotationMatrix3D(pi/4, pi/5, 0);
Usage:

Rotations::Public Vars

Function:	<ul style="list-style-type: none">□ <code>double alpha;</code>□ <code>double beta;</code>□ <code>double chi;</code>
Input:	<ul style="list-style-type: none">□
Description:	<ul style="list-style-type: none">□ These are the current Spinner angles. These are set when 'setPowderAngles' is called.<ul style="list-style-type: none">alpha is typically {time}*{spinning speed}beta is typically the rotor angle (i.e. the MagicAngle)chi is almost always 0.
Example Usage:	<ul style="list-style-type: none">□ <code>Rotations myRots(Rotations::Spherical);</code>□ <code>myRots.setRotorAngles(0.1*5000*PI2, 54.7*DEG2RAD);</code>□ <code>cout<<myRots.beta;</code>

Rotations::Public Vars

Function: **rmatrix** **cartPowder**

Input:

Description: This holds the Cartesian Rotation matrix (a real 3x3 matrix) for rotation of vectors for the Powder Angles. It is calculated is the RotationType=All, or Cartesian everytime 'setPowderAngles' is called.

Example Rotations myRots(Rotations::Cartesian);

Usage: myRots.setPowderAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);
coord<> v(1,1,1); //a vector to rotate
v=myRots.cartPowder*v;//rotate the vector

Rotations::Public Vars

Function: **rmatrix** **cartSpin;**

Input:

Description: This holds the Cartesian Rotation matrix (a real 3x3 matrix) for rotation of vectors for the Rotor Angles. It is calculated is the RotationType=All, or Cartesian everytime 'setPowderAngles' is called.

Example Rotations myRots(Rotations::Cartesian);

Usage: myRots.setRotorAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);
coord<> v(1,1,1); //a vector to rotate
v=myRots.cartSpin*v; //rotate the vector

Rotations::Public Vars

Function:

```
□ double phi;  
double theta;  
double gamma;
```

Input:

□

Description:

□ These are the current Powder angles. These are set when 'setPowderAngles' is called.

Example Usage:

□

Rotations::Public Vars

Function: `matrix powderAs;`

Input: `matrix`

Description: `matrix` This variable stores the entire Wigner rank 2 matrix (a 5x5 Full matrix). It is calculated is the RotationType=All, or Spherical everytime 'setPowderAngles' is called.

Example `Rotations myRots(Rotations::Spherical);`

Usage: `myRots.setPowderAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);
myRots.powderAs(2,2); //get the wigner(m=0, mp=0) component`

Rotations::Public Vars

Function: **int RotationType;**

Input:

Description: This stores the selected type of rotations to perform. Its values can be

Rotations::Spherical

Rotations::Cartesian

Rotations::All

If Rotations::Spherical is selected, then the rotational components 'powderAs' and 'spinnerAs' will be calculated using the Spherical Tenors upon a new input of Euler angles to either 'setPowderAngles' or 'setRotorAngles'.

If Rotations::Cartesian is selected, then the rotational components 'cartPowder' and 'cartSpin' will be calculated using the Cartesian Rotation matrices upon a new input of Euler angles to either 'setPowderAngles' or 'setRotorAngles'.

If Rotations::All is selected, then both Spherical and Cartesian variables will be calculated. THIS IS THE DEFAULT.

Example **Rotations myRots(Rotations::Cartesian);**
Usage: **//change the calculation type to spherical**
 myRots.RotationType=Rotations::Spherical;

Rotations::Public Vars

Function: `Vector<complex> spinnerAs;`

Input:

Description: This variable stores the entire Wigner rank 2 values for m=0, mp=-2, -1, 0, 1, 2 in a length 5 Vector. It is calculated is the RotationType=All, or Spherical everytime 'setPowderAngles' is called.

Example `Rotations myRots(Rotations::Spherical);`

Usage: `myRots.setRotorAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);`
 `myRots.spinnerAs(2); //get the wigner(m=0, mp=0) component`

Rotations::constructor

Function: **Rotations(int type=Rotations::All);**

Input: type--> a Rotation Type integer

Description: This is a constructor that requires a rotation type. The type can be

Rotations::Spherical

Rotations::Cartesian

Rotations::All

If Rotations::Spherical is selected, then the rotational components 'powderAs' and 'spinnerAs' will be calculated using the Spherical Tenors upon a new input of Euler angles to either 'setPowderAngles' or 'setRotorAngles'.

If Rotations::Cartesian is selected, then the rotational components 'cartPowder' and 'cartSpin' will be calculated using the Cartesian Rotation matrices upon a new input of Euler angles to either 'setPowderAngles' or 'setRotorAngles'.

If Rotations::All is selected, then both Spherical and Cartesian variables will be calculated. THIS IS THE DEFAULT.

Example **Rotations AllRots; //the default is Rotations::All**

Usage: **Rotations myRots(Rotations::Cartesian);**

Rotations::other

Function: **void setPowderAngles(double phi, double theta, double gamma=0.);**

Input: [alpha, beta, gamma]--> Euler angles in RADIANS

Description: Sets the rotation variables for the powder rotations. It will only calculate the variables IF the angles 'alpha, beta, gamma' have changed since the function was last called. It will also choose to calculate the variables based on the RotationType.

Example Rotations myRots(Rotations::Cartesian);

Usage: myRots.setPowderAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);

Rotations::other

Function: **void setRotorAngles(double alpha, double beta, double gamma=0.0);**

Input: [alpha, beta, gamma]--> Euler angles in RADIANS

Description: Sets the rotation variables for the rotor rotations. It will only calculate the variables IF the angles 'alpha, beta, gamma' have changed since the function was last called. It will also choose to calculate the variables based on the RotationType.

Example Rotations myRots(Rotations::Cartesian);

Usage: myRots.setRotorAngles(45*DEG2RAD, 45*DEG2RAD, 45*DEG2RAD);

Using the spacial and spin tensors and a Rotations object to generate the Zero-Field dipolar hamiltonian.

This is a class demonstration that uses the spacial and spherical tensors to create the 2 functions nessesary for the driver class 'oneFID' to calculate FIDs from a specific hamiltonian embeded in a class.

```
#include "blochlib.h"

//need to use the proper namespaces
using namespace BlochLib;
using namespace std;

//a class that extends the SolidSys
// nessesary in order for the 'oneFID' to function
// properly. This class overwrites the 'Hamiltonian'
// functions of SolidSys
class ZeroFieldDipole :
public SolidSys
{
public:
double D;
ZeroFieldDipole():SolidSys(), D(0){}
ZeroFieldDipole(double d):SolidSys(), D(d){}
ZeroFieldDipole(SolidSys &sys, double d):SolidSys(sys), D(d){}

//This function (and the one below) are nessesary for the
// oneFID driver to calculate the FIDs from
// these hamiltonians
hmatrix Hamiltonian(double t1, double t2, double wr)
{
//the object 'theRotations' is a Rotations object
// embedded inside the SolidSys base class
    double tmid=((t2-t1)/2.0+t1)*wr*PI2);
    return D*(
A2(tmid,theRotations.beta,-2)*T22(*this, 0,1)
-A2(tmid,theRotations.beta,-1)*T21(*this, 0,1)
+A2(tmid,theRotations.beta,0)*T20(*this, 0,1)
-A2(tmid,theRotations.beta,1)*T2m1(*this, 0,1)
-A2(tmid,theRotations.beta,2)*T2m2(*this, 0,1));
}

hmatrix Hamiltonian(double wr, double rot, double alpha, double beta, double t1, double t2)
{
return sqrt(6.0)*D*(
A2(alpha,beta,-2)*T22(*this, 0,1)
-A2(alpha,beta,-1)*T21(*this, 0,1)
+A2(alpha,beta,0)*T20(*this, 0,1)
-A2(alpha,beta,1)*T2m1(*this, 0,1)
+A2(alpha,beta,2)*T2m2(*this, 0,1));
}

};
```

--Global Functions class Csa { ...

--- read_csa

--Public Vars

--- T00, T20

--Constructors

--- Csa

--- Csa

--- Csa

Chemical Shift Anisotropy this class holds all the relevant data for csa and calculates the Hamiltonian optimized for each rotation type.

--Element Extraction

--- alpha, beta, gamma

--- getParam

--- iso, del, eta

--- si, chi, psi

--- spinON, on

--Assignments

--- alpha, beta, gamma

--- iso, del, eta

--- on

--- operator=

--- setParam

--- si, psi, chi

--IO

--- display

--- operator<<

--- write

--Other Functions

--- ==, !=

--- H, Hamiltonian

--- is_zero

--- setCrystalAs

--- setSpinMats

Examples

--- simple usage

Csa::global function

Function: **Vector<Csa> read_csa(Vector<std::string> &in)**

Input: in--> A vector of strings that typically comes from reading in a file (but not necessarily).

Description: Read the string

C {iso} {del} {eta} {on} {si} {chi} {psi}

from an element in the Vector<std::string>.

Example Usage: data.push_back("C 89 900 0.1 0");
data.push_back("C 4455 900 0.1 1");
data.push_back("D 500 0 1");

Vector<Csa> mycsa=rea_csa(data);
// mycsa has 2 csa in it...

Csa::Public Vars

Function: **rимatrix T00;**
 matrix T20;

Input:

Description: T20 --> the second rank m=0 spin matrix
T00 --> the isotopic tensor (Iz)

These are the spin tensors for that spin in the ENTIRE Hilbert Space of a spin system. Therefore in order for this class to correctly function, you must perform a 'setSpinMats' functional call, before you can calculate hamiltonians.

Example SpinSys A(3);
Usage: //the csa is ready... to be used
Csa myc(iso, del, eta, spin);
myc.setSpinMats(sys);
cout<<myc.T00<<endl;

Csa::constructor

Function:

- **Csa()**

Input:

- void

Description:

- empty constructor

DEFAULTS:::

iso=0.;

del=0.;

eta=0.;

si=0.;

chi=0.;

psi=0.;

chain=0;

Example Usage:

- ```
Csa moo;
cout<<moo.iso(); //prints out '0'
```

## Csa::constructor

**Function:**  `Csa(double iso, double del, double eta, int spin, double si=0, double chi=0, double psi=0)`

**Input:**  iso--> isotropic shift  
del--> anisotropic shift  
eta--> assymetry parameter (0..1)  
spin--> the spin number inside the 'SpinSys'  
[si, chi, psi]--> Euler angle for relative molecular orientation

**Description:**  creates a Csa data structure

**Example**  `Csa moo(500, 2333, .3, 2);`

**Usage:** `cout<<moo.iso()<<" "<<moo.del(); //prints '500 2333'`

## Csa::constructor

### Function:

□ **Csa(const Csa &cp)**

### Input:

□ cp--> an existing SpinSys to copy..

### Description:

□ creates a copy of the old Csa...COPIES SPIN-OPS ALSO

### Example Usage:

□ `Csa moo(230, 3000, 0, 1);  
Csa loo(moo); //moo is copied into loo`

## Csa::element extraction

### Function:

- `double alpha();`
- `double beta();`
- `double gamma();`

### Input:

- `void`

### Description:

- These are the same as [si, psi, chi] for the Euler angles [alpha, beta, gamma].

### Example Usage:

- `Csa moo(230, 3000, 0, 1);`
- `cout<<moo.alpha(); //prints '0' same as moo.si()`

## Csa::element extraction

### Function:

□ **double getParam(std::string which)**

### Input:

□ which-->The info flag you wish to get (which can be  
"iso", "del", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on"

### Description:

□ Returns the parameter corresponding to the input string.  
All 'int' types are converted to doubles.

which can be

"iso", "del", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on"

### Example Usage:

□ `Csa moo(230, 3000, 0, 1);  
cout<<moo.getParam("iso"); //prints '230'`

## Csa::element extraction

### Function:

- double iso();**
- double del();**
- double eta();**

### Input:

- void

Description:  Returns the isotropic shift (iso), anisotropy (del), and asymmetry (eta) parameters.

Example Usage:  Csa moo(230, 3000, 0, 1);  
cout<<moo.del(); //prints '3000'

## Csa::element extraction

### Function:

- `double si();`
- `double psi();`
- `double chi();`

### Input:

- `void`

### Description:

- Returns the Euler angles [si, psi, chi].

### Example Usage:

- `Csa moo(230, 3000, 0, 1);`
- `cout<<moo.si(); //prints '0'`

## Csa::element extraction

### Function:

- `int spinOn();`
- `int on();`
- `void`
- Returns spin number label.
- `Csa moo(230, 3000, 0, 1);`
- `cout<<moo.on(); //prints '1'`

### Input:

### Description:

### Example Usage:

## Csa::assignments

**Function:** □ `void alpha(double in);`  
              `void beta(double in);`  
              `void gamma(double in);`

**Input:** □ `in-->` the new value for the [si, psi, chi] angle (DEGREES)

**Description:** □ These functions set the Euler orientation angles [alpha, beta, gamma]. These are equivalent to the [si, psi, chi]versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

**Example** □ `Csa moo(230, 3000, 0, 1);`

**Usage:**     `cout<<moo.beta(); //prints '0'`  
              `moo.beta(35); //changes to 35 degrees..`

## Csa::assignments

- Function:**
- `void iso(double in);`
  - `void del(double in);`
  - `void eta(double in)`
- Input:**
- `in`--> the new value for the iso, del, or eta
- Description:**
- Allows the setting of the isotropic shift (iso), anisotropy (del), and asymmetry (eta) parameters.
- Example Usage:**
- `Csa moo(230, 3000, 0, 1);`
  - `cout<<moo.iso(); //prints '230'`
  - `moo.iso(600); //changes to 600`

## Csa::assignments

### Function:

□ **int on(int in)**

### Input:

□ in--> the new value for the spin index

### Description:

□ Allows the setting of spin number label.

### Example Usage:

□ `Csa moo(230, 3000, 0, 1);  
cout<<moo.on(); //prints '1'  
moo.on(2); //changes to 2`

## Csa::assignments

**Function:**  **Csa &operator=(const Csa &rhs);**

**Input:**  rhs--> the item you wish to copy

**Description:**  Assigns one Csa from another.  
This also copies all the Spin Operators if they have been already calculated.

**Example Usage:**  Csa moo(230, 3000, 0, 1);  
Csa koo;  
koo=moo;

## Csa::assignments

### Function:

□ **void setParam(std::string which, double val);**

### Input:

□ which--> the item you wish to set..

"iso", "del", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on"

val--> that value you wish to set it to...

### Description:

□ Sets a value inside the structure (any 'int' will be converted properly.)

### Example Usage:

□ Csa moo(230, 3000, 0, 1);

moo.setParam("iso", 56); //cahnge iso to 56

## Csa::assignments

**Function:** □ `void si(double in);`  
□ `void psi(double in);`  
□ `void chi(double in);`

Input: □ in--> the new value for the [si, psi, chi] angle (DEGREES)

Description: □ These functions set the Euler orientation angles [si, psi, chi]. These are equivalent to the alpha, beta, gamma versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

Example □ `Csa moo(230, 3000, 0, 1);`

Usage: `cout<<moo.chi(); //prints '0'`

`moo.chi(35); //changes to 35 degrees..`

## Csa::IO

**Function:**  **void display()**

**Input:**  void

**Description:**  Writes the string

```
#-----CSA-----
spin iso del eta alpha beta gamma
{on} {iso} {del} {eta} {si} {chi} {psi}
```

to the Console.

**Example Usage:**  Csa moo(230, 3000, 0, 1);

```
moo.display();
/*
will print out...
#-----CSA-----
spin iso del eta alpha beta gamma
1 230 3000 0 0 0 0
*/
```

## Csa::IO

### Function:

□ **ostream &operator<<(stream &oo, Csa &out)**

### Input:

□ oo--> an output string  
out--> a Csa object

### Description:

□ Writes the string....  
#-----CSA-----  
# spin iso del eta alpha beta gamma  
{on} {iso} {del} {eta} {si} {chi} {psi}

### Example Usage:

□ Csa moo(230, 3000, 0, 1);  
cout<<moo;  
/\*  
will print out...  
#-----CSA-----  
# spin iso del eta alpha beta gamma  
1 230 3000 0 0 0 0  
\*/

## Csa::IO

**Function:** `void write(ostream &out)`

**Input:** `out-->` file/output stream you wish to write a VALID, READABLE string for the 'read\_csa' function

**Description:** `Writes the string....`

`C {iso} {del} {eta} {on} {si} {chi} {psi}`

to the ostream.

This string is then readable by the 'read\_csa' function below to produce a valid Csa object.

**Example** `Csa moo(230, 3000, 0, 1);`

**Usage:** `ostream out( "params" );  
moo.write(out);`

`// C 230 3000 0 1 0 0 0`

## Csa::other

### Function:

- `bool operator==(const Csa &rhs);`
- `bool operator!=(const Csa &rhs);`

### Input:

- `rhs-->` another Csa

### Description:

- Determines whether or not the rhs is the same Csa as the lhs.

### Example Usage:

- `Csa moo(230, 3000, 0, 1), loo(500, 3000, 0, 1);`
- `moo==loo; //would be false`
- `moo.iso()=500;`
- `moo==loo; //true`

## Csa::other

**Function:**

- `dmatrix H(SpinSys &sys, Rotations &rot)`
- `dmatrix Hamiltonian(SpinSys &sys, Rotations &rot)`
  
- `dmatrix H(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`
- `dmatrix Hamiltonian(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`

**Input:**

- sys--> the Master Spin System
- rot--> the 'Rotations' class structure (see 'spin/space tensor')

alpha--> the rotor Phase (usually wr\*time\*PI2) (RADIANS)  
beta--> the rotor angle (RADIANS)  
theta--> theta powder angle (RADIANS)  
phi--> phi powder angle (RADIANS)

**Description:**

- The master Hamiltonian function returns the Hamiltonian in matrix form...at the proper angles

The bottom two functions are typically meant for Liquid type sims. The one that explicitly include the 'Rotations' class are faster only in that they do not need to calculate a 'Rotations' each iteration.

`H=A_Csa_pas(iso, 0,0,0,0)*T00+(some rotation stuff)*T20`

**Example**

- `Csa moo(230, 3000, 0, 1);`
- `SpinSys sys(2);`
- `moo.setSpinMats(sys);`
- `Rotations rots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle`
- `cout<<moo.H(sys, rots)<<endl; // valid hamiltonian`

**Usage:**

## Csa::other

### Function:

□ **bool is\_zero()**

### Input:

□ **void**

### Description:

□ Determines weather or not the interacting is 'zero'  
i.e. if iso==0 and del==0 then the interaction is usless.

### Example Usage:

□ `Csa moo(230, 3000, 0, 1), loo(230, 3000, 0, 1);  
moo.is_zero(); // false`

## Csa::other

**Function:**  **void setCrystalAs()**

Input:  void

Description:  Sets the necessary initial crystal A2m's from the [si, chi, psi] and [iso, del, eta] set of parameters.

This is done automatically at initialization and whenever you change ANY parameter (except for 'on').

Example  Csa moo( 230, 3000, 0, 1);

Usage: SpinSys sys(2);  
moo.setSpinMats(sys);  
Rotations rots(34\*Pi/180, 56\*Pi/180); // [phi theta] powder angle  
cout<<moo.H(sys, rots)<<end; // valid hamiltonian

## Csa::other

**Function:** `void setSpinMats(SpinSys &sys)`

**Input:** `sys-->` A vlid spin system

**Description:** `Set the 'T20' and 'T00' spin matricies from the spin system.`

**Example Usage:**

```
Csa moo(230, 3000, 0, 1);
SpinSys sys(2);
moo.setSpinMats(sys);
cout<<moo.T20<<end;
```

## Simple usage of the Csa class

---

This example shows how to declare and set the parameters, as well as using the Rotations class to get the CSA Hamiltonian.

---

```
double iso=4000, del=3000, eta=0, spin=1;
SpinSys sys(2);
//the csa is ready... to be used
Csa myc(iso, del, eta, spin); //need to set the correct spin matrices before calling the hamiltonian
myc.setSpinMats(sys);
Rotations Rots;
Rots.setPowderAngles(56*Pi/180, 70*Pi/180);
//the hamiltonian...at the powder angle (56, 70);
cout<<myc.H(sys, Rots);
```

---

--Global Functions    class Qua { ...

---

--- read\_qua

--Public Vars              Quadrupoles  
--- T20, T21T2m1,  
T22T2m2, c2, c4              Holds all the relevant data for a quadrupole  
Calulates the hamiltonian optimized for each rotation type.  
Both first and second order quadrupoles are included in the Hamiltonian  
To turn OFF the second order factors simply set the 'Bfield' parameter to 0.

---

--Constructors  
--- Qua

---

--- Qua

---

--- Qua

---

--Element Extraction

--- alpha, beta, gamma

--- Bfield

--- getParam

--- Q, eta

--- si, chi, psi

--- spinON, on

---

--Assignments

--- alpha, beta, gamma

--- Bfield

--- on

--- operator=

--- Q, eta

--- setParam

--- si, psi, chi

---

--IO

--- display

--- operator<<

--- write

---

--Other Functions

--- ==, !=

--- H, Hamiltonian

--- is\_zero

--- setCrystalAs

--- setSpinMats

Examples

---

--- simple usage

## Quadrupole::global function

**Function:** □ `Vector<Qua> read_qua(Vector<std::string> &in)`

**Input:** □ in--> A vector of strings that typically comes from reading in a file (but not necessarily).

**Description:** □ Read the string

Q {Q} {eta} {on} {si} {chi} {psi} {order}

from an element in the Vector<std::string>.

**Example Usage:** □ `data.push_back( "C 89 900 0.1 0" );`  
  `data.push_back( "C 4455 900 0.1 1" );`  
  `data.push_back( "D 500 0 1" );`  
  `data.push_back( "Q 500e5 0 1" );`  
  
  `Vector<Qua> myqua=rea_qua(data);`  
  `// myqua has 1 qua in it...`

## Quadrupole::Public Vars

**Function:**  **rimatrix** T00;  
**matrix** T21T2m1;  
**matrix** T22T2m2;  
**rdmatrix** c2;  
**rdmatrix** c4;

**Input:**

**Description:**  T20 --> the second rank m=0 spin matrix  
T21T2m1 --> a second order spin matrix  
T22T2m2 --> a second order spin matrix

c2--> the rank 2 total second order spin matrix (combined space)  
c4--> the rank 4 total second order spin matrix (combined space)

These are the spin tensors for that spin in the ENTIRE Hilbert Space of a spin system. Therefore in order for this class to correctly function, you must perform a 'setSpinMats' functional call, before you can calculate hamiltonians.

**Example**  SpinSys A(3);

**Usage:**   
//the qua is ready... to be used  
Qua myc(Q, eta, spin);  
myc.setSpinMats(sys);  
cout<<myc.T20<<endl;

## Quadrupole::constructor

### Function:

- **Qua( )**

### Input:

- void

### Description:

- empty constructor

DEFAULTS:::

Q=0.;

eta=0.;

si=0.;

chi=0.;

psi=0.;

on1=0;

on2=0;

order=1;

Bfield=0;

### Example Usage:

- ```
Qua moo;
cout<<moo.Q(); //prints out '0'
```

Quadrupole::constructor

Function: `Qua(double Q, double eta, int spin, double si=0, double chi=0, double psi=0, Bfield=0.0)`

Input: `Q`--> anisotropic Quadrupolar coupling
`eta`--> assymetry parameter (0..1)
`spin`--> the spin number inside the 'SpinSys'
`[si, chi, psi]`--> Euler angle for relative molecular orientation
`Bfield`--> the Magnetic Field strength in Hz (nesseary for proper calculationg of the second order quadrupole.)

Description: creates a Qua data structure

Example `Qua moo(50e6, .3, 2);`

Usage: `cout<<moo.Q()<<" "<<moo.eta(); //prints 5e7 0.3'`

Quadrupole::constructor

Function:	<code>Qua(const Qua &cp)</code>
Input:	<code>cp--></code> an existing SpinSys to copy..
Description:	creates a copy of the old Qua...COPIES SPIN-OPS ALSO
Example Usage:	<code>Qua moo(230, 3000, 0, 1);</code> <code>Qua loo(moo); //moo is copied into loo</code>

Quadrupole::element extraction

Function:

- `double alpha();`
- `double beta();`
- `double gamma();`

Input:

- `void`

Description:

- These are the same as [si, psi, chi] for the Euler angles [alpha, beta, gamma].

Example Usage:

- `Qua moo(230, 3000, 0, 1);`
- `cout<<moo.alpha(); //prints '0' same as moo.si()`

Quadrupole::element extraction

Function: **double Bfield()**

Input: void

Description: Returns the Magnetic Field strength in Hz.

Example Usage: Qua moo(3e6, 0, 1);
cout<<moo.Bfield(); //prints '0'
moo.Bfield(35.e6); //changes to 35 MHz

Quadrupole::element extraction

- Function:** `double getParam(std::string which)`
- Input:** `which-->The info flag you wish to get (which can be "Q", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on", "Bfield", "order"`
- Description:** `which` can be
"Returns the parameter corresponding to the input string.
All 'int' types are converted to doubles.
- `"iso", "del", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on"`
- Example Usage:** `Qua moo(230, 0.1, 1);
cout<<moo.getParam("Q"); //prints '230'`

Quadrupole::element extraction

Function:

- `double Q();`
- `double eta();`

Input:

- `void`

Description:

- Returns the coupling strength (Q) and asymmetry (eta) parameters.

Example Usage:

- `Qua moo(230e6, 0.3, 1);`
- `cout<<moo.Q(); //prints '2.3e8'`

Quadrupole::element extraction

Function:

- `double si();`
- `double psi();`
- `double chi();`

Input:

- `void`

Description:

- Returns the Euler angles [si, psi, chi].

Example Usage:

- `Qua moo(230, 3000, 0, 1);`
- `cout<<moo.si(); //prints '0'`

Quadrupole::element extraction

Function:

- `int spinOn();`
- `int on();`

Input:

- `void`

Description:

- Returns spin number label.

Example Usage:

- `Qua moo(230, 3000, 0, 1);`
- `cout<<moo.on(); //prints '1'`

Quadrupole::assignments

Function:

- `void alpha(double in);`
- `void beta(double in);`
- `void gamma(double in);`

Input:

- `in-->` the new value for the [si, psi, chi] angle (DEGREES)

Description:

- These functions set the Euler orientation angles [alpha, beta, gamma]. These are equivalent to the [si, psi, chi]versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

Example

- `Qua moo(230, 3000, 0, 1);`

Usage:

- `cout<<moo.beta(); //prints '0'`

- `moo.beta(35); //changes to 35 degrees..`

Quadrupole::assignments

Function:

□ **void Bfield(double in)**

Input:

□ in--> the new value for the Bfield in Hz

Description:

□ Sets the magnetic field strength (needed for second order quadrupole)

Example Usage:

```
□ Qua moo(3e6, 0, 1);
  cout<<moo.Bfield(); //prints '0'
  moo.Bfield(35.e6); //changes to 35 MHz
```

Quadrupole::assignments

Function:

□ **int on(int in)**

Input:

□ in--> the new value for the spin index

Description:

□ Allows the setting of spin number label.

Example Usage:

□ `Qua moo(230, 3000, 0, 1);
cout<<moo.on(); //prints '1'
moo.on(2); //changes to 2`

Quadrupole::assignments

Function: `Qua &operator=(const Qua &rhs);`

Input: `rhs-->` the item you wish to copy

Description: `Assigns one Qua from another.`
This also copies all the Spin Operators if they have been already calculated.

Example Usage: `Qua moo(230, 3000, 0, 1);`

`Qua koo;`

`koo=moo;`

Quadrupole::assignments

Function:

- `void Q(double in);`
- `void eta(double in)`

Input:

- `in-->` the new value for the Q or eta

Description:

- Sets the coupling strength (Q) and asymmetry (eta) parameters.

Example Usage:

- `Qua moo(230, 3000, 0, 1);`
- `cout<<moo.iso(); //prints '230'`
- `moo.iso(600); //changes to 600`

Quadrupole::assignments

Function: `void setParam(std::string which, double val);`

Input:

- which--> the item you wish to set..
"Q", "eta", "si", "chi", "psi", "alpha", "beta", "gamma", "on", "Bfield", "order"

- val--> that value you wish to set it to...

Description: □ Sets a value inside the structure (any 'int' will be converted properly.)

Example Usage: □ `Qua moo(230, 0, 1);`
`moo.setParam("Bfield", 560e6); //changes Bfield to 560 MHz`

Quadrupole::assignments

Function:

- `void si(double in);`
- `void psi(double in);`
- `void chi(double in);`

Input:

- `in-->` the new value for the [si, psi, chi] angle (DEGREES)

Description:

- These functions set the Euler orientation angles [si, psi, chi]. These are equivalent to the alpha, beta, gamma versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

Example

- `Qua moo(230, 3000, 0, 1);`

Usage:

- `cout<<moo.chi(); //prints '0'`

- `moo.chi(35); //changes to 35 degrees..`

Quadrupole::IO

Function: **void display()**

Input: void

Description: Writes the string

```
#-----Quad-----  
# spin iso del eta alpha beta gamma  
{on} {Q} {eta} {si} {chi} {psi}
```

to the Console.

Example Usage: Qua moo(230, 3000, 0, 1);

```
moo.display();  
/*  
will print out...  
#-----CSA-----  
# spin iso del eta alpha beta gamma  
1 230 3000 0 0 0 0  
*/
```

Quadrupole::IO

Function: `ostream &operator<<(stream &oo, Qua &out)`

Input: `oo-->` an output string
`out-->` a Qua object

Description: `Writes the string....`
`#-----Quad-----`
`# spin del eta alpha beta gamma order`
`{on} {Q} {eta} {si} {chi} {psi} {order}`

Example Usage: `Qua moo(230, 0.1, 0, 1);`
`cout<<moo;`
`/*`
`will print out...`
`#-----QUAD-----`
`# spin Q eta alpha beta gamma order`
`1 230 0.1 0 0 0 1`
`*/`

Quadrupole::IO

Function: `void write(ostream &out)`

Input: `out-->` file/output stream you wish to write a VALID, READABLE string for the 'read_qua' function

Description: `Writes the string....`

`Q {Q} {eta} {on} {si} {chi} {psi} {order}`

to the ostream.

This string is then readable by the 'read_qua' function below to produce a valid Qua object.

Example `Qua moo(230, 0, 1);`

Usage: `ostream out("params");
moo.write(out);`

`// Q 230 0 1 0 0 0 1`

Quadrupole::other

Function:

- `bool operator==(const Qua &rhs);`
- `bool operator!=(const Qua &rhs);`

Input:

- `rhs->` another Qua

Description:

- Determines whether or not the rhs is the same Qua as the lhs.

Example Usage:

```
□ Qua moo(230, 0, 1), loo(500, 0, 1);
moo==loo; //would be false
moo.Q(500);
moo==loo; //true
```

Quadrupole::other

Function:

- `dmatrix H(SpinSys &sys, Rotations &rot)`
- `dmatrix Hamiltonian(SpinSys &sys, Rotations &rot)`

- `dmatrix H(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`
- `dmatrix Hamiltonian(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`

Input:

- sys--> the Master Spin System
- rot--> the 'Rotations' class structure (see 'spin/space tensor')

- alpha--> the rotor Phase (usually wr*time*PI2) (RADIANS)
- beta--> the rotor angle (RADIANS)
- theta--> theta powder angle (RADIANS)
- phi--> phi powder angle (RADIANS)

Description:

- The master Hamiltonian function returns the Hamiltonian in matrix form...at the proper angles

The bottom two functions are typically meant for Liquid type sims. The one that explicitly include the 'Rotations' class are faster only in that they do not need to calculate a 'Rotations' each iteration.

```
let Qp=Q/(4*qn()*(2*qn()-1));  
H1=Qp*(some rotation stuff)*T20  
H2=Qp*({some rotation stuff}*c4-{some rotation stuff}*c2)  
Htot=H1+H2
```

Example

- Qua moo(230, 3000, 0, 1);

Usage:

- `SpinSys sys(2);`
- `moo.setSpinMats(sys);`
- `Rotations rots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle`
- `cout<<moo.H(sys, rrots)<<endl; //valid hamiltonian`

Quadrupole::other

Function:	<code>bool is_zero()</code>
Input:	<code>void</code>
Description:	<ul style="list-style-type: none">□ Determines weather or not the interacting is 'zero' i.e. if $Q==0$ then the interaction is usless.
Example Usage:	<ul style="list-style-type: none">□ <code>Qua moo(230, 0, 1);</code> <code>moo.is_zero(); // false</code>

Quadrupole::other

Function: **void setCrystalAs()**

Input: void

Description: Sets the necessary initial crystal A2m's from the [si, chi, psi] and [iso, del, eta] set of parameters.

This is done automatically at initialization and whenever you change ANY parameter (except for 'on').

Example Qua moo(230, 0, 1);

Usage: SpinSys sys(2);
moo.setSpinMats(sys);
Rotations rrots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle
cout<<moo.H(sys, rrots)<<endl; // valid hamiltonian

Quadrupole::other

Function: **void setSpinMats(SpinSys &sys)**

Input: sys--> A valid spin system

Description: Sets the T20, T21T2m1, T22T2m2, c2, and c4 spin matrices from the spin system.

Example Usage: Qua moo(230, 0, 1);
SpinSys sys(2);
moo.setSpinMats(sys);
cout<<moo.T20<<end;

Simple usage of the Quad class

This example shows how to declare and set the parameters, as well as using the Rotations class to get the Quadrupole Hamiltonian.

```
double Q=3e5, eta=0, spin=1;
SpinSys sys(2);
sys(spin)="14N"; //set a spin to the quadrupole
//the Qua is ready... to be used
Qua myc(Q, eta, spin);

//need to set the correct spin matrices
// before calling the hamiltonian
myc.setSpinMats(sys);
Rotations Rots;
Rots.setPowderAngles(56*Pi/180, 70*Pi/180);

//the hamiltonian...at the powder angle (56, 70);
cout<<myc.H(sys, Rots);
//to get the second order Quad you need
//to set the Magnetic field to a value other then 0
myc.Bfield(400e6); //400 MHz bfield

//the hamiltonian...at the powder angle (56, 70)
//with second order Secular terms included
cout<<myc.H(sys, Rots);
```

--Global Functions class Dip { ...

--- read_dip

--Public Vars Dipole-Dipole interactions. This clas holds all the relevent data for dipole
and calulates the hamiltonian optimized for each rotation type.

--- T20

--Constructors

--- Dip

--- Dip

--- Dip

--Element Extraction

--- alpha, beta, gamma

--- del

--- getParam

--- si, chi, psi

--- spinON1, on1

--- spinON2, on2

--Assignments

--- alpha, beta, gamma

--- del

--- on1, on2

--- operator=

--- setParam

--- si, psi, chi

--IO

--- display

--- operator<<

--- write

--Other Functions

--- ==, !=

--- H, Hamiltonian

--- is_zero

--- setCrystalAs

--- setSpinMats

Examples

--- simple usage

Dipole-Dipole::global function

Function: `□ Vector<Dip> read_dip(Vector<std::string> &in)`

Input: `□ in--> A vector of strings that typically comes from reading in a file (but not necessarily).`

Description: `□ Read the string`

`D {del} {on1} {on2} {si} {chi} {psi}`

`from an element in the Vector<std::string>.`

Example Usage: `□ data.push_back("C 89 900 0.1 0");
data.push_back("C 4455 900 0.1 1");
data.push_back("D 500 0 1");

Vector<Dip> mydip=re_dip(data);
// mydip has 1 dip in it...`

Dipole-Dipole::Public Vars

Function: **matrix T20;**

Input:

Description: T20 --> the second rank m=0 spin matrix

These are the spin tensors for that spin in the ENTIRE Hilbert Space of a spin system. Therefore in order for this class to correctly function, you must perform a 'setSpinMats' functional call, before you can calculate hamiltonians.

Example Spinsys A(3);

Usage: //the dip is ready... to be used
Dip myc(del, spin1, spin2);
myc.setSpinMats(sys);
cout<<myc.T00<<endl;

Dipole-Dipole::constructor

Function:

- Dip()**

Input:

- void

Description:

- empty constructor
DEFAULTS:::
 del=0.;
 si=0.;
 chi=0.;
 psi=0.;
 spin1=0;
 spin2=0;

Example Usage:

- `Dip moo;`
`cout<<moo.del(); //prints out '0'`

Dipole-Dipole::constructor

Function: `Dip(double del, int spin1,int spin2, double si=0, double chi=0, double psi=0)`

Input: `del--> anisotropic shift (dipole-dipole coupling)`
`spin1--> the first spin number inside the 'SpinSys'`
`spin2--> the second spin number inside the 'SpinSys'`
`[si, chi, psi]--> Euler angle for relative molecular orientation`

Description: `creates a Dip data structure`

Example `Dip moo(500, 0, 2);`

Usage: `cout<<moo.del(); //prints '500'`

Dipole-Dipole::constructor

Function:

□ **Dip(const Dip &cp)**

Input:

□ cp--> an existing SpinSys to copy..

Description:

□ creates a copy of the old Dip...COPIES SPIN-OPS ALSO

Example Usage:

□ `Dip moo(3000, 0, 1);
Dip loo(moo); //moo is copied into loo`

Dipole-Dipole::element extraction

Function:

- `double alpha();`
- `double beta();`
- `double gamma();`

Input:

- `void`

Description:

- These are the same as [si, psi, chi] for the Euler angles [alpha, beta, gamma].

Example Usage:

- `Dip moo(3000, 0, 1);`
- `cout<<moo.alpha(); //prints '0' same as moo.si()`

Dipole-Dipole::element extraction

Function:

□ **double del();**

Input:

□ void

Description:

□ Returns the anisotropy (del the dipole coupling) parameter.

Example Usage:

□

```
Dip moo(3000, 0, 1);
cout<<moo.del(); //prints '3000'
```

Dipole-Dipole::element extraction

Function:

□ **double getParam(std::string which)**

Input:

□ which-->The info flag you wish to get (which can be
"del", "si", "chi", "psi", "alpha", "beta", "gamma", "on1", "on2"

Description:

□ Returns the parameter corresponding to the input string.
All 'int' types are converted to doubles.

which can be

"del", "si", "chi", "psi", "alpha", "beta", "gamma", "on1", "on2"

Example Usage:

□ `Dip moo(3000, 0, 1);
cout<<moo.getParam("del")); //prints '3000'`

Dipole-Dipole::element extraction

Function:

- `double si();`
- `double psi();`
- `double chi();`

Input:

- `void`

Description:

- Returns the Euler angles [si, psi, chi].

Example Usage:

- `Dip moo(3000, 0, 1);`
- `cout<<moo.si(); //prints '0'`

Dipole-Dipole::element extraction

Function:

`int spinOn1();`
 `int on1();`

Input:

`void`

Description:

Returns FIRST spin number label.

Example Usage:

`Dip moo(3000, 0, 1);`
`cout<<moo.on1(); //prints '0'`

Dipole-Dipole::element extraction

Function:

- `int spinOn2();`
- `int on2();`

Input:

- `void`

Description:

- Returns SECOND spin number label.

Example Usage:

- `Dip moo(3000, 0, 1);`
- `cout<<moo.on2(); //prints '1'`

Dipole-Dipole::assignments

Function:

- `void alpha(double in);`
- `void beta(double in);`
- `void gamma(double in);`

Input:

- `in-->` the new value for the [si, psi, chi] angle (DEGREES)

Description:

- These functions set the Euler orientation angles [alpha, beta, gamma]. These are equivalent to the [si, psi, chi]versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

Example

- `Dip moo(3000, 0, 1);`

Usage:

- `cout<<moo.beta(); //prints '0'`

- `moo.beta(35); //changes to 35 degrees..`

Dipole-Dipole::assignments

Function: □ **void del(double in);**

Input: □ in--> the new value for the del

Description: □ Allows the setting of the anisotropy (del the dipole coupling) parameter.

Example Usage: □ Dip moo(230, 0, 1);
 cout<<moo.del(); //prints '230'
 moo.del(600); //changes to 600

Dipole-Dipole::assignments

Function: `void on1(int in);`
`void on2(int in)`

Input: `in--> the new value for the spin index`

Description: Allows the setting of FIRST (on1) spin number label and the second (on2) number label.

Example Usage: `Dip moo(3000, 0, 1);`
`cout<<moo.on1(); //prints '1'`
`moo.on1(2); //changes to 2`
`moo.on2(1); //changes to 1`

Dipole-Dipole::assignments

Function: `Dip &operator=(const Dip &rhs);`

Input: `rhs-->` the item you wish to copy

Description: `Assigns one Dip from another.`
This also copies all the Spin Operators if they have been already calculated.

Example Usage: `Dip moo(230, 0, 1);`
`Dip koo;`
`koo=moo;`

Dipole-Dipole::assignments

Function:

□ **void setParam(std::string which, double val);**

Input:

□ which--> the item you wish to set..
"del", "si", "chi", "psi", "alpha", "beta", "gamma", "on1", "on2"

val--> that value you wish to set it to...

Description:

□ Sets a value inside the structure (any 'int' will be converted properly.)

Example Usage:

□ Dip moo(230, 0, 1);
moo.setParam("del", 56); //cahnges del to 56

Dipole-Dipole::assignments

Function:

- `void si(double in);`
- `void psi(double in);`
- `void chi(double in);`

Input:

- `in--> the new value for the [si, psi, chi] angle (DEGREES)`

Description:

- These functions set the Euler orientation angles [si, psi, chi]. These are equivalent to the alpha, beta, gamma versions of these functions. Upon calling these functions, the crystal Orientation tensors components are recalculated.

Example

- `Dip moo(3000, 0, 1);`

Usage:

- `cout<<moo.chi(); //prints '0'`

- `moo.chi(35); //changes to 35 degrees..`

Dipole-Dipole::IO

Function: **void display()**

Input: void

Description: Writes the string

```
#-----Dipd-----
# spin del alpha beta gamma
{on1},{on2} {del} {si} {chi} {psi}
```

to the Console.

Example Usage: Dip moo(230, 3000, 0, 1);

```
moo.display();
/*
will print out...
#-----DIP-----
# spin del alpha beta gamma
0,1 230 0 0 0
*/
```

Dipole-Dipole::IO

Function: **ostream &operator<<(stream &oo, Dip &out)**

Input: oo--> an output string
 out--> a Dip object

Description: Writes the string....
-----Dipd-----
spin del alpha beta gamma
{on1},{on2} {del} {si} {chi} {psi}

Example Usage: Dip moo(230, 0, 1);
cout<<moo;
/*
will print out...
-----DIP-----
spin del alpha beta gamma
0,1 230 0 0 0
*/

Dipole-Dipole::IO

Function: `void write(ostream &out)`

Input: `out-->` file/output stream you wish to write a VALID, READABLE string for the 'read_dip' function

Description: `Writes the string....`

`D {del} {on1} {on2} {si} {chi} {psi}`

to the ostream.

This string is then readable by the 'read_dip' function below to produce a valid Dip object.

Example `Dip moo(230, 0, 1);`

Usage: `ostream out("params");
moo.write(out);`

`// D 230 0 1 0 0 0`

Dipole-Dipole::other

Function:

- `bool operator==(const Dip &rhs);`
- `bool operator!=(const Dip &rhs);`

Input:

- `rhs--> another Dip`

Description:

- Determines weather or not the rhs is the not the same Dip as the lhs.

Example Usage:

- `Dip moo(230, 3000, 0, 1), loo(500, 3000, 0, 1);`
- `moo==loo; //would be false`
- `moo.del(500);`
- `moo==loo; //true`

Dipole-Dipole::other

Function:

- `dmatrix H(SpinSys &sys, Rotations &rot)`
- `dmatrix Hamiltonian(SpinSys &sys, Rotations &rot)`

- `dmatrix H(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`
- `dmatrix Hamiltonian(SpinSys &sys, double alpha=0, double beta=0, double theta=0, double phi=0)`

Input:

- sys--> the Master Spin System
- rot--> the 'Rotations' class structure (see 'spin/space tensor')

- alpha--> the rotor Phase (usually wr*time*PI2) (RADIANS)
- beta--> the rotor angle (RADIANS)
- theta--> theta powder angle (RADIANS)
- phi--> phi powder angle (RADIANS)

Description:

- The master Hamiltonian function returns the Hamiltonian in matrix form...at the proper angles

The bottom two functions are typically meant for Liquid type sims. The one that explicitly include the 'Rotations' class are faster only in that they do not need to calculate a 'Rotations' each iteration.

`H={some rotation stuff}*T20`

Example

- `Dip moo(230, 0, 1);`
- `SpinSys sys(2);`
- `moo.setSpinMats(sys);`
- `Rotations rots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle`
- `cout<<moo.H(sys, rots)<<endl; // valid hamiltonian`

Usage:

Dipole-Dipole::other

Function:

- **bool is_zero()**

Input:

- void

Description:

- Determines weather or not the interacting is 'zero'
i.e. if del==0 then the interaction is usless.

Example Usage:

- ```
Dip moo(3000, 0, 1);
moo.is_zero(); // false
```

## Dipole-Dipole::other

**Function:**  **void setCrystalAs()**

Input:  void

Description:  Sets the necessary initial crystal A2m's from the [si, chi, psi] and [iso, del, eta] set of parameters.

This is done automatically at initialization and when ever you change ANY parameter (except for 'on').

Example  Dip moo(230, 0, 1);

Usage: SpinSys sys(2);

```
moo.setSpinMats(sys);
Rotations rrots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle
cout<<moo.H(sys, rrots)<<end; // valid hamiltonian
```

## Dipole-Dipole::other

### Function:

□ **void setSpinMats(SpinSys &sys)**

### Input:

□ sys--> A valid spin system

### Description:

□ Sets the 'T20' spin matrices from the spin system.

### Example Usage:

```
□ Dip moo(3000, 0, 1);
 SpinSys sys(2);
 moo.setSpinMats(sys);
 cout<<moo.T20<<end;
```

This example shows how to declare and set the parameters, as well as using the Rotations class to get the Quadrupole Hamiltonian.

---

```
double del=3000;
int spin1=0, spin2=1;
SpinSys sys(2);
Dip moo(del, spin1, spin2); //the Dip is ready... to be used
myc.setSpinMats(sys); //need to set the correct spin matrices before calling the hamiltonian
Rotations Rots;
Rots.set_powderAs(56*Pi/180, 70*Pi/180);
cout<<myc.H(sys, Rots); //the hamiltonian...at the powder angle (56, 70);
```

---

--Global Functions class J{ ...

--- read\_js

--Public Vars

--- T00

Scalar, J, Coupling This holds all the relevant data for J and calculates the hamiltonian.

---

--Constructors

--- J

--- J

--- J

--Element Extraction

--- getParam

--- iso

--- spinON1, on1

--- spinON2, on2

--Assignments

--- iso

--- on1, on2

--- operator=

--- setParam

--IO

--- display

--- operator<<

--- write

--Other Functions

--- ==, !=

--- H, Hamiltonian

--- is\_zero

--- setSpinMats

Examples

--- simple usage

## J-coupling::global function

**Function:**  `Vector<J> read_js(Vector<std::string> &in)`

**Input:**  `in`--> A vector of strings that typically comes from reading in a file (but not necessarily).

**Description:**  Read the string

J {iso} {on1} {on2} {strong}

from an element in the `Vector<std::string>`.

**Example Usage:**  `data.push_back( "C 89 900 0.1 0" );`  
`data.push_back( "C 4455 900 0.1 1" );`  
`data.push_back( "J 500 0 1 1" );`  
  
`Vector<J> myj=read_js(data);`  
`// myj has 1 j in it...`

## J-coupling::Public Vars

**Function:**  **matrix T00;**

Input:

Description:  T00 --> The spin tensos (either Iz\*Iz or I.I depending on the strong tag).

These are the spin tensors for that spin in the ENTIRE Hilbert Space of a spin system. Therefore in order for this class to correctly function, you must perform a 'setSpinMats' functional call, before you can calculate hamiltonians.

Example  Spinsys A(3);

Usage:  //the j is ready... to be used  
J myc(iso, spin1, spin2);  
myc.setSpinMats(sys);  
cout<<myc.T00<<endl;

## J-coupling::constructor

### Function:

- J()**

### Input:

- void

### Description:

- empty constructor

DEFAULTS:::

iso=0.;

si=0.;

chi=0.;

psi=0.;

spin1=0;

spin2=0;

### Example Usage:

- `J moo;`  
`cout<<moo.iso(); //prints out '0'`

## J-coupling::constructor

### Function:

□ `J( double iso, int spin1,int spin2,int strong=0)`

### Input:

- iso--> anisotropic shift (jole-jole coupling)
- spin1--> the first spin number inside the 'SpinSys'
- spin2--> the second spin number inside the 'SpinSys'
- strong--> is the coupling a 'strong' (1) or 'weak' (0) coupling

### Description:

- creates a J data structure

### Example Usage:

□ `J moo(500, 0, 2);  
cout<<moo.iso(); //prints '500'`

## J-coupling::constructor

### Function:

□ **J(const J &cp)**

### Input:

□ cp--> an existing SpinSys to copy..

### Description:

□ creates a copy of the old J...COPIES SPIN-OPS ALSO

### Example Usage:

□ `J moo(3000, 0, 1);  
J loo(moo); //moo is copied into loo`

## J-coupling::element extraction

### Function:

□ **double getParam(std::string which)**

### Input:

□ which-->The info flag you wish to get (which can be  
"iso", "on1", "on2", "strong"

### Description:

□ Returns the parameter corresponding to the input string.  
All 'int' types are converted to doubles.

which can be  
"iso", "on1", "on2", "strong"

### Example Usage:

□ `J moo(3000, 0, 1);  
cout<<moo.getParam("iso"); //prints '3000'`

## J-coupling::element extraction

### Function:

□ **double iso();**

### Input:

□ **void**

### Description:

□ Returns the anisotropy (iso the jole coupling) parameter.

### Example Usage:

□ **J moo(3000, 0, 1);  
cout<<moo.iso(); //prints '3000'**

## J-coupling::element extraction

### Function:

- `int spinOn1();`
- `int on1();`

### Input:

- `void`

### Description:

- Returns FIRST spin number label.

### Example Usage:

- `J moo( 3000, 0, 1);`
- `cout<<moo.on1(); //prints '0'`

## J-coupling::element extraction

### Function:

- `int spinOn2();`
- `int on2();`

### Input:

- `void`

### Description:

- Returns SECOND spin number label.

### Example Usage:

- `J moo( 3000, 0, 1);`
- `cout<<moo.on2(); //prints '1'`

## J-coupling::assignments

### Function:

□ **void iso(double in);**

### Input:

□ in--> the new value for the iso

### Description:

□ Allows the setting of the anisotropy (iso the jole coupling) parameter.

### Example Usage:

□ `J moo(230, 0, 1);  
cout<<moo.iso(); //prints '230'  
moo.iso(600); //changes to 600`

## J-coupling::assignments

**Function:**  `void on1(int in);`  
`void on2(int in)`

**Input:**  `in--> the new value for the spin index`

**Description:**  Allows the setting of FIRST (on1) spin number label and the second (on2) number label.

**Example Usage:**  `J moo(3000, 0, 1);`  
`cout<<moo.on1(); //prints '1'`  
`moo.on1(2); //changes to 2`  
`moo.on2(1); //changes to 1`

## J-coupling::assignments

**Function:** `J &operator=(const J &rhs);`

**Input:** `rhs-->` the item you wish to copy

**Description:** `Assigns one J from another.`  
`This also copies all the Spin Operators if they have been already calculated.`

**Example Usage:** `J moo(230, 0, 1);`  
`J koo;`  
`koo=moo;`

## J-coupling::assignments

**Function:** `void setParam(std::string which, double val);`

**Input:** `which--> the item you wish to set..  
"iso","on1", "on2", "strong"`

`val--> that value you wish to set it to...`

**Description:** `Sets a value inside the structure (any 'int' will be converted properly.)`

**Example Usage:** `J moo(230, 0, 1);  
moo.setParam("iso", 56); //cahnges iso to 56`

## J-coupling::IO

**Function:**  **void display()**

**Input:**  void

**Description:**  Writes the string

```
#-----J-----
spin iso
{on1},{on2} {iso}
```

to the Console.

**Example Usage:**  J moo(230, 0, 1);
moo.display();
/\*
will print out...
#-----J-----
# spin iso
0,1 230
\*/

## J-coupling::IO

**Function:** `ostream &operator<<(stream &oo, J &out)`

**Input:** `oo-->` an output string  
`out-->` a J object

**Description:** `oo` writes the string....  
`#-----J-----`  
`# spin iso`  
`{on1},{on2} {iso}`

**Example Usage:** `J moo(230, 0, 1);`  
`cout<<moo;`  
`/*`  
`will print out...`  
`#-----J-----`  
`# spin iso`  
`0,1 230`  
`*/`

## J-coupling::IO

**Function:** `void write(ostream &out)`

**Input:** `out-->` file/output stream you wish to write a VALID, READABLE string for the 'read\_j' function

**Description:** `Writes the string....`

`D {iso} {on1} {on2} {si} {chi} {psi}`

to the ostream.

This string is then readable by the 'read\_j' function below to produce a valid J object.

**Example** `J moo(230, 0, 1);`

**Usage:** `ostream out("params");  
moo.write(out);`

`// D 230 0 1 0 0 0`

## J-coupling::other

### Function:

- `bool operator==(const J &rhs);`
- `bool operator!=(const J &rhs);`

### Input:

- `rhs--> another J`

### Description:

- Determines weather or not the rhs is the not the same J as the lhs.

### Example Usage:

```
□ J moo(230, 0, 1), loo(500, 0, 1);
moo==loo; //would be false
moo.iso(500);
moo==loo; //true
```

## J-coupling::other

**Function:**

- `hamtrix H(SpinSys &A)`  
`hamtrix Hamiltonian(SPinSys &A);`
- `dmatrix H(SpinSys &sys, Rotations &rot)`  
`dmatrix Hamiltonian(SpinSys &sys, Rotations &rot)`
- `dmatrix H(SpinSys &sys,double alpha=0, double beta=0, double theta=0, double phi=0)`  
`dmatrix Hamiltonian(SpinSys &sys,double alpha=0, double beta=0, double theta=0, double phi=0)`

**Input:**

- sys--> the Master Spin System
- rot--> the 'Rotations' class structure (see 'spin/space tensor')

- alpha--> the rotor Phase (usually wr\*time\*PI2) (RADIANS)
- beta--> the rotor angle (RADIANS)
- theta--> theta powder angle (RADIANS)
- phi--> phi powder angle (RADIANS)

**Description:**

- The master Hamiltonian function returns the Hamiltonian in matrix form...at the proper angles

The bottom two functions are typically meant for Liquid type sims. The one that explicitly include the 'Rotations' class are faster only in that they do not need to calculate a 'Rotations' each iteration.

H=iso\*T00

**Example**

- `J moo(230, 0, 1);`
- `SpinSys sys(2);`
- `moo.setSpinMats(sys);`
- `Rotations rots(34*Pi/180, 56*Pi/180); // [phi theta] powder angle`
- `cout<<moo.H(sys, rots)<<endl; // valid hamiltonian`

**Usage:**

## J-coupling::other

### Function:

- **bool is\_zero()**

### Input:

- **void**

### Description:

- Determines weather or not the interacting is 'zero'  
i.e. if iso==0 then the interaction is usless.

### Example Usage:

- **J moo( 3000, 0, 1);  
moo.is\_zero(); // false**

## J-coupling::other

### Function:

□ **void setSpinMats(SpinSys &sys)**

### Input:

□ sys--> A valid spin system

### Description:

□ Sets the 'T00' spin matrices from the spin system.

### Example Usage:

```
□ J moo(3000, 0, 1);
 SpinSys sys(2);
 moo.setSpinMats(sys);
 cout<<moo.T00<<end;
```

This example shows how to declare and set the parameters, as how to get the J-coupling Hamiltonian.

---

```
double iso=3000;
int strong=0, spin1=0, spin2=1;
SpinSys sys(2);
//the J is ready... to be used
J myc(iso, spin1, spin2, strong); //need to set the correct spin matrices before calling the hamiltonian
myc.setSpinMats(sys);
Rotations Rots;
Rots.setPowderAngles(56*Pi/180, 70*Pi/180);
//the hamiltonian...at the powder angle (56, 70);
cout<<myc.H(sys, Rots);
//this is the same as the above
cout<<myc.H(sys);
```

---

**--Constructors**[--- HamiltonianGen](#)[--- HamiltonianGen](#)**--Other Functions**[--- Hamiltonian](#)[Examples](#)[--- FIDs](#)

class HamiltonianGen { ...

Generates hamiltonians from an input std::string like  
 "2000\*A22\*T22\_0 ,1" used for generation of detection matrices, initial density operators, or any other spin function you wish to know from an input string.

Below You will find the available functions

The constants `pi`, `e`, and `complexi=(complex(0,1))` are available. Even though no direction use of 'complex(#,#)' can be used, the `complexi` will create complex numbers.

You are now allowed to perform basic functions on either the numbers or the matrices. NOTE:: all functions on the matrices are Element-By-Element...for example, the expression 'exp(Ix)' will NOT return the matrix exponential, but is equivalent to `exp(Ix(i,j))` for each element in the matrix.

The available math functions included so far are

```
exp, ln, log, sin, cos, tan, asin, acos,
atan, sinh, cosh, tanh, asinh, acosh,
atanh, abs, floor, ceil, sqrt, real,
imag
```

Available functional pieces

- Spatial spherical tensors:  
 rank 1: **A11, A10, A1m1**  
 rank 2: **A22, A21, A20, A2m1, A2m2**
- Spin spherical tensors:  
 rank 1: **T11, T10, T1m1**  
 rank 2: **T22,T21,T20,T2m1,T2m2**
- Pauli Spin Matrices:  
**Ix, Iy, Iz, Ii, Ip, Im**
- plain numbers (sorry no complex numbers at this time, except 'complexi')

--Selecting spin tensors and pauli matrices for a specific spin

--IF no '\_#' is present the total spin tensor is used for pauli matrices however, you will get an error if you use the rank 2 spin tensors (and the spin space is larger than 2) without specifying the spins.

- Special Tensors:: Not spin specific
- Spin Tensors:  
 rank 1 (single spin tensors):  
**T11\_0** (means T11 of spin 0)  
**T1m1\_2** (means T1-1 for spin 2)

rank 2 (double spin tensors):

**T20\_1,2** (means T20 from spin 1 and 2)

**T2m1\_0,2** (means T2-1 from spin 0 and 2)

rank 2 (same spin tensors:: make sure you have defined a spin with s>1/2 or these will be zero...i.e. a quadrupole)

**T20\_1,1** (means T20 for spin 1)

- Pauli Matrices:

**Ix\_1** (means Ix from spin 1)

**Ip\_4** (means I+ from spin 4)

Operators:: EVERY thing except the first and last element require the normal operators...'+', '-', '/', '\*'

Example functions:: Input Strings

500\*A20\*T20\_0,1

4000\*(A21\*T21\_1,0+A2m1\*T2m1\_1,0)

A22\*T22\_0,1+A20\*T20\_1,1+Ix\_0+Iz\_1

---

## HamiltonianGen::constructor

**Function:**       **HamiltonianGen( );**

**Input:**             void

**Description:**         The empty constructor  
                        simply declares an object ready to be used for Hamiltonian Generation.

**Example Usage:**     HamiltonianGen myGen;

## HamiltonianGen::constructor

**Function:** `HamiltonianGen(std::string func);`

**Input:** `func-->` the input string to parse (like "A22\*T20")

**Description:** `func-->` Declares an object ready to be used for Hamiltonian Generation, and sets the function string to parse later by 'Hamiltonian.'

**Example** `HamiltonianGen myGen( "Iy_0*Iy_1" );`

**Usage:** `SolidSys A( 2 );`

`//generates 'Iy_0*Iy_1' from the`

`// total hilbert space of A`

`matrix myMat=myGen.Hamiltonian(A);`

## HamiltonianGen::other

**Function:**  **matrix** Hamiltonian(**SpinSys** &A, double theta=0, double phi=0);  
 **matrix** Hamiltonian(**SpinSys** &A, std::string func, double theta=0, double pho=0);

**Input:**  A--> a SolidSys or a SpinSys (or any class derived from them)  
func--> a valid function string (like Ix\_0)  
theta--> the powder THETA angle in RADIANS  
phi--> the powder PHI angle in RADIANS

**Description:**  Generates the matrix from a string 'func' (or if the string constructor was called, from that input string). You only need to enter in 'theta' and 'phi' if there are Spacial Tensors in the input string.

**Example**  HamiltonianGen myGen;

**Usage:**  SolidSys A(2);  
A[0] = "1H";  
A[1] = "13C";  
//returns the Ix matrix from the first spin in  
// the TOTAL Hilbert Space of A  
matrix myMat = myGen.Hamiltonian(A, "Ix\_0");

## Collect FIDs based on an input Hamiltonian String

---

This program generates FID from an input Hamiltonian. It takes in a Parameter Set file like so

-----in.sim-----

```
#a simple parameter input file for the HamilGen
fid maker
```

```
npts 512
sw 20000
aveType zcw
thetaSteps 616
phiSteps 233
```

```
numspins 2
```

```
#simple dipole
hamil 2000*A20*T20_01
```

```
detect Ip
roeq Ix
```

-----End in.sim-----

---

```
#include "blochlib.h"

using namespace BlochLib;
using namespace std;

//This extends the SolidSys class
// and overwrites the two
// Hamiltonian Functions required by
// the 'oneFID' object..
//it simply returns the matrix generated from
// the input hamiltonian
class HamilSys: public SolidSys
{
public:
 std::string hamil;
 HamiltonianGen myGen;

 HamilSys(int nspins, std::string Hamil=""):
 SolidSys(nspins),
 hamil(Hamil),
 myGen(Hamil)
 {}

 hmatrix Hamiltonian(double t1, double t2, double wr=0.0)
 {
 SpinSys & tm=(SpinSys &)(*this); //cast a ptr back to SPinSys
 return myGen.Hamiltonian(tm, hamil, theRotations.theta, theRotations.phi);
 }

 hmatrix Hamiltonian(double wr, double rot, double alpha, double beta, double t1, double t2)
 {
 SpinSys & tm=(SpinSys &)(*this); //cast a ptr back to SPinSys
 return myGen.Hamiltonian(tm, hamil, alpha, beta);
 }
};

int main(int argc,char* argv[]){

 //start MPI
 MPIworld.start(argc, argv);

 std::string inf="";
 if(MPIworld.master()) query_parameter(argc,argv,1, "InputFile:: ", inf);
```

```

MPIworld.scatter(inf);

//make a parameter set
Parameters pset(inf);
//decalare our 'HamilSys'
HamilSys mysys(
 pset.getParamI("numspins"),
 pset.getParamS("hamil")
);

//nnum points in fid
int npts=pset.getParamI("npts");
//the sweep width
double sw=pset.getParamD("sw");
//our detection matrix
std::string detectST=pset.getParamS("detect");
//our starting matrix
std::string roeqST=pset.getParamS("roeq");
//powder average type
std::string aveType=pset.getParamS("aveType");
int thetaStep=pset.getParamI("thetaSteps", "", false);
int phiStep=pset.getParamI("phiSteps", "", false);

//decalre our powder
powder zcw(aveType, thetaStep, phiStep);

//using this hamiltonianGen to create the
//detect and roeq matrices
HamiltonianGen mygen;
matrix detect=mygen.Hamiltonian(mysys, detectST);
matrix roeq=mygen.Hamiltonian(mysys, roeqST);

//our fid vector
Vector<complex> fid(npts, 0);
//decalre an 'oneFID' object
oneFID<HamilSys> myfid(mysys, npts, sw);

//set the MPI controller of the oneFID
myfid.Controller=MPIworld;

//collect the FID
fid=myfid.FID(zcw, roeq, detect);
complex sm=0;
//dc offset correct it
sm=sum(fid(Range(npts/2, npts)))/double(npts/2.0);
fid-=sm;
//print out the data
if(MPIworld.master()) plotterFID(fid, "fid", 1.0/myfid.sw());

//end MPI
MPIworld.end();
}

```

---

**--Public Vars**

--- csa  
--- dip  
--- jcop  
--- qua

--- theRotations

**--Constructors**

--- SolidSys  
--- SolidSys  
--- SolidSys  
--- SolidSys

**--Element Extraction**

--- Bfield, Bo

**--Assignments**

--- addCsa  
--- addDip  
--- addJ  
--- addQ  
--- setBfield

**--IO**

--- display  
--- operator<<  
--- print  
--- read  
--- read  
--- write  
--- write

**--Other Functions**

--- Hamiltonian  
--- isDiagonal  
--- setCrystalAs  
--- setPowderAngles  
--- setRotorAngles  
--- setSpinMats  
--- size

class SolidSys : public SpinSys { ...

---

This class integrates both the SpinSys (Spin operators and parameters) and the interactions (Dip, Csa, J, Qua). It maintains a Rotations object as well as lists of each interaction to create total system Hamiltonians.

---

## SolidSys::Public Vars

|                       |                                                                                                                                                                                   |
|-----------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Function:</b>      | <input type="checkbox"/> <code>Vector&lt;Csa&gt; csa;</code>                                                                                                                      |
| <b>Input:</b>         | <input type="checkbox"/>                                                                                                                                                          |
| <b>Description:</b>   | <input type="checkbox"/> Holds the list of CSAs in the SolidSys.                                                                                                                  |
| <b>Example Usage:</b> | <input type="checkbox"/><br><code>SolidSys moo( 3 );</code><br><code>// a solid system with 1 csa on the 2nd spin</code><br><code>moo.csa.push_back(Csa(200, 4000, 0, 1));</code> |

## SolidSys::Public Vars

### Function:

□ `Vector<Dip> dip;`

### Input:

□

### Description:

□ Holds the list of Dipoles in the SolidSys.

### Example Usage:

```
□ SolidSys moo(3);
 //a solid system with 1 dipole
 // between the first and second spin
moo.csa.push_back(Dip(200, 0, 1));
```

## SolidSys::Public Vars

**Function:**  **Vector<J>** jcop;

**Input:**

**Description:**  Holds the list of J-couplings in the SolidSys.

**Example Usage:**  SolidSys moo(3);  
//a solid system with 1 csa on the 2nd spin  
moo.jcoup.push\_back(J(200, 0, 1));

## SolidSys::Public Vars

**Function:**  `Vector<Qua> qua;`

**Input:**

**Description:**  Holds the list of Quadrupoles in the SolidSys.

**Example Usage:**  `SolidSys moo( 3 );`  
`//a solid system with 1 csa on the 2nd spin`  
`moo[ 0 ] = "2H"; //a quadrupole`  
`moo.qua.push_back( Qua( 20e6, 0.1, 0 ) );`

## SolidSys::Public Vars

**Function:**  **Rotations theRotations**

Input:

Description:  This object iss the master rotations object that is used to calculate the total Hamiltonians of the system from each interaction.

Example  SolidSys moo(3);

Usage: moo.theRotations.setPowderAngles(PI/3, PI/3);

## SolidSys::constructor

**Function:**  **SolidSys(SolidSys &cp)**

**Input:**  cp--> the SolidSys you wish to copy

**Description:**  The copy constructor. Copies all relevant information (theRotations, Bfield, the interactions, etc)

**Example Usage:**  SolidSys moo( 4 );  
SolidSys loo(moo);

## SolidSys::constructor

**Function:** `□ SolidSys(SpinSys &sys)`

**Input:** `□ sys-->` a previously declared SpinSys

**Description:** `□` Sets up an object from a previously declared SpinSys object. The 'Bfield' is set to 0 and all angles in 'theRotations' are set to 0.

**Example** `□ SpinSys sys(3); //3 spin system`

**Usage:** `SolidSys moo(sys); //a solid system`

## SolidSys::constructor

**Function:** `□ SolidSys(int numsp)`

**Input:** `□ numsp-->` the number of spins in the system

**Description:** `□` Sets up an object with 'numsp' spins, all protons. There are no interactions however, and the Bfield is set to 0.

**Example** `□ SolidSys moo(3); //3 spin 1/2 protons`

**Usage:**

## SolidSys::constructor

**Function:**  **SolidSys()**

**Input:**  void

**Description:**  empty constructor. No interactions are set and Bfield is set to 0. There is only one spin in the system, a proton.

**Example**  SolidSys moo;  
**Usage:**

## SolidSys::element extraction

### Function:

- double Bfield();**
- double Bo();**

### Input:

- void

### Description:

- Returns the Bfield for the spin system.

### Example Usage:

- ```
SolidSys moo(3);
moo.setBfield(300e6);
cout<<moo.Bfield();
```

SolidSys::assignments

Function: `void addCsa(Csa c);`

Input: `c-->` A new CSA to add to the system

Description: Adds a CSA to Solid System and sets the Spin Operators for the new csa.

Example Usage:

```
SolidSys moo( 3 );
//adds a csa
moo.addCsa(Csa(200, 0.1,1));
```

SolidSys::assignments

Function: `void addDip(Dip d);`

Input: `d-->` A new Dipole to add to the system

Description: Adds a Dipole to Solid System and sets the Spin Operators for the new dipole.

Example Usage: `SolidSys moo(3);
//adds a dipole
moo.addDip(Dip(200, 0,1));`

SolidSys::assignments

Function:

□ **void addJ(J inj);**

Input:

□ inj--> a J coupling object

Description:

□ Adds a J-coupling to the Solid System

Example Usage:

```
□ SolidSys moo( 3 );
  //adds a j-coupling
  moo.addJ(J( 200, 0,1 ));
```

SolidSys::assignments

Function: `void addQ(Qua q);`

Input: `q-->` a quadrupole object

Description: Adds a Quadrupole to Solid System, sets the Bfield, and sets the Spin Operators for the new quadrupole.

Example `SolidSys moo(3);
Usage: //adds a quadrupole
 moo.addQ(Qua(200, 0.3,1));`

SolidSys::assignments

Function: **void setBfield(double Bf);**

Input: Bf-> the magnetic field in Hz

Description: Sets the Bfield of the system AND all the Bfield of any quadrupoles in the system.

Example SolidSys moo(3);

Usage: moo[0] = "14N";

moo.setBfield(300e6);

moo.addQ(Qua(3000, 0, 0)); //this set the Bfield from the SolidSys

SolidSys::IO

Function: `void display();`

Input: `void`

Description: `is the same as 'print(std::cout)' print the string`

```
#-----
#-----spin parameters-----
#-----
#you have {numspins} spins in your system
```

...A line for each Interaction in the system...

Example Usage: `SolidSys A(3);
A.csa.push_back(Csa(400, 0, 0, 1));
A.display();
//will print a nice formatted string to the console`

SolidSys::IO

Function: `□ std::ostream &operator<<(std::ostream &otr, SolidSys &out);`

Input: `□ otr--> an ostream`
`out--> the SolidSys object you wish to print`

Description: `□ The output printing operator. Performs the same task as 'print(otr).'`

Example Usage: `□ SolidSys A(3);`
`A.csa.push_back(Csa(400, 0, 0, 1));`
`cout<<A<<endl;`
`//will print a nice formatted string to the console`

SolidSys::IO

Function: `void print(std::ostream &oo)`

Input: `oo--> an ostream`

Description: `Prints a nice formatted string to the ostream oo of the form`

```
#-----
#-----spin parameters-----
#-----
#you have {numspins} spins in your system
```

...A line for each Interaction in the system...

Example Usage: `SolidSys A(3);
A.csa.push_back(Csa(400, 0, 0, 1));
A.print(std::cout);
//will print a nice formatted string to the console`

SolidSys::IO

Function: **void read(const std::string in);**
 void read(const char * in);

Input: in--> a file name

Description: This reads the SolidSys from a file by the name 'in.'

It reads this format

```
numspins {num}  
T {label} {spin count}  
C {iso} {del} {eta} {spin} {si} {psi} {chi}  
D {del} {spin1} {spin2} {si} {psi} {chi}  
J {iso} {spin1} {spin2} {si} {psi} {chi} {strong}  
Q {Q} {eta} {spin} {si} {psi} {chi} {order}
```

Example Usage: //suppose our input file was

```
// numspins 2  
// T 1H 0  
// T 13C 1  
// C 200 300 0.1 0  
// C 0 4000 0 1  
// D 400 0 1  
  
//declare our spin system  
SolidSys moo;  
moo.read("infile");  
  
//the size of SolidSys is 2, with 2 CSAs and 1 dipole
```

SolidSys::IO

Function: `■ read(Vector<std::string> list);`

Input: `■ list--> a Vector<std::string>`

Description: `■` Reads a Solid Sys from a Vector<std::string> with each entry being the same line as from an input file 'read' function.

Example `■ Vector<std::string> list;`

Usage:

`----//suppose our input file was`
`list.push_back("numspins 2");`
`list.push_back("T 1H 0");`
`list.push_back("T 13C 1");`
`list.push_back(" C 200 300 0.1 0");`
`list.push_back("C 0 4000 0 1");`
`list.push_back("D 400 0 1");`

`----//suppose our input file was`
`// numspins 2`
`// T 1H 0`
`// T 13C 1`
`// C 200 300 0.1 0`
`// C 0 4000 0 1`
`// D 400 0 1`

`//declare our spin system`
`SolidSys moo;`
`moo.read(list);`

`//the size of SolidSys is 2, with 2 CSAs and 1 dipole`

SolidSys::IO

Function: `void write(std::ostream &otr);`

Input: `otr-->` an ostream

Description: `Writes a SolidSys READABLE (via 'read') chunk of text. It has the form`

```
numspins {num}
T {label} {spin count}
C {iso} {del} {eta} {spin} {si} {psi} {chi}
D {del} {spin1} {spin2} {si} {psi} {chi}
J {iso} {spin1} {spin2} {si} {psi} {chi} {strong}
Q {Q} {eta} {spin} {si} {psi} {chi} {order}
```

depending on what is in the SolidSys

Example Usage: `// suppose our input file was`

```
// numspins 2
// T 1H 0
// T 13C 1
// C 200 300 0.1 0
// C 0 4000 0 1
// D 400 0 1

//declare our spin system
SolidSys moo;
moo.read("infile");
//the size of SolidSys is 2, with 2 CSAs and 1 dipole

moo.csa[0].iso(3000);
ostream out("outfile");
moo.write(out);
//will print this to the 'outfile'
// numspins 2
// T 1H 0
// T 13C 1
// C 3000 300 0.1 0
// C 0 4000 0 1
// D 400 0 1
```

SolidSys::IO

Function: `void write(std::string &otr);`

Input:

- `otr-->` the output file name, the file will be OVERWRITTEN with this data

Description:

- Writes a SolidSys READABLE (via 'read') chunk of text. It has the form

```
numspins {num}
T {label} {spin count}
C {iso} {del} {eta} {spin} {si} {psi} {chi}
D {del} {spin1} {spin2} {si} {psi} {chi}
J {iso} {spin1} {spin2} {si} {psi} {chi} {strong}
Q {Q} {eta} {spin} {si} {psi} {chi} {order}
```

depending on what is in the SolidSys

Example Usage: `// suppose our input file was`

```
// numspins 2
// T 1H 0
// T 13C 1
// C 200 300 0.1 0
// C 0 4000 0 1
// D 400 0 1

//declare our spin system
SolidSys moo;
moo.read("infile");
//the size of SolidSys is 2, with 2 CSAs and 1 dipole

moo.csa[0].iso(3000);
moo.write("outfile");
//will print this to the 'outfile'
// numspins 2
// T 1H 0
// T 13C 1
// C 3000 300 0.1 0
// C 0 4000 0 1
// D 400 0 1
```

SolidSys::other

Function:

- `hmatrix &Hamiltonian(double wr , double beta, double th, double phi, double t1=0, double t2=0)`
- `hmatrix &Hamiltonian(double t1, double t2, double wr=0.0);`
- `hmatrix &Hamiltonian();`

Input:

- `wr`--> the spinning speed current phase (typically $wr * PI2 * t$) in RADIANS
- `beta`--> the rotor angle (typically the magic angle) in RADIANS
- `th`--> the powder THETA angle in RADIANS
- `phi`--> the powder PHI angle in RADIANS
- `t1`--> the begining time for the 'dt' step
- `t2`--> the end time for the 'dt' step

Description:

- This returns the total Hamiltonian for the entire spin system with all the interactions, given the angles and/or the 'dt' time step. It returns a REFERENCE to an internal private member. This is for SPEED, as the matrix needs to only be initialized once.

The best and fastest way to use this function is AFTER calling 'setPowderAngles' and 'setRotorAngles' as the 'theRotations' object gets updated appropriately. If you call these two functions, then 'Hamiltonian()' is the same as 'Hamiltonian(t1, t2)'

Example

```
□ SolidSys A(3);
A.csa.push_back(Csa(400, 0, 0, 1));
A.setSpinMats();
A.setCrystalAs();
A.setPowderAngles(pi/3, pi/4, pi/5);
double wr=5000.0, t1=0.1, t2=0.15, ma=acos(1.0/sqrt(3.0));
A.setRotorAngles(wr*((t2-t1)/2.0+t1)*PI2,ma);
cout<<A.Hamiltonian(t1, t2,wr);
cout<<A.Hamiltonian(); //same as above
```

Usage:

SolidSys::other

Function: `bool isDiagonal()`

Input: `void`

Description: If the object only contains CSA and Quadrupole interactions (i.e. the size of jcop and dip are 0) then the generated hamiltonian is a diagonal real matrix and this function will return true.

Example `SolidSys A(2);`

Usage: `A.addCsa(Csa(300, 2000, 0,1));
A.isDiagonal(); //this is true
A.addDip(Dip(2000, 0, 1));
A.isDiagonal(); //this is false`

SolidSys::other

Function: **void setCrystalAs();**

Input: void

Description: This sets all the Cyrstal Spacial Tensors (the relative orientations of each interaction) for each interaction in the SolidSys. If you add an interaction via 'mySys.csa.push_back(...)' then you must call these two functions afterwards to get correct hamiltonans 'setSpinMats()' and 'setCrystalAs()'. Using the functions 'addCsa', etc. will automatically call these functions, however, if you plan to add many interactions, the 'addCsa', etc functions are much slower. It is better to use the 'push_back' functions for each Interaction list and THEN call the two function 'setCrystalAs()' and 'setSpinMats()'

Example SolidSys A(3);

Usage: A.csa.push_back(Csa(400, 0, 0, 1));
 A.setSpinMats();
 A.setCrystalAs();

SolidSys::other

Function: □ **void setPowderAngles(double theta, double phi, double gam=0.0);**

Input: □ theta--> the theta powder angle in RADIANS

phi--> the phi powder angle in RADIANS

gamma--> the gamma powder angle in RADIANS

Description: □ This sets the powder angles in 'theRotations' objects. This is equivalent to

SolidSys::theRotations.setPowderAngles(phi, theta, gamma)

Example □ SolidSys moo(3);

Usage: moo.setPowderAngles(pi/3, pi/5);

//this is the same as above, but note the order change

moo.theRotations.setPowderAngles(pi/5, pi/3);

SolidSys::other

Function: □ **void setRotorAngles(double wr, double beta, double chi=0)**

Input: □ wr--> the alpha rotor angle in RADIANS (usually wr*t*PI2)
beta--> the beta rotor angle in RADIANS
chi--> the gamma powder angle in RADIANS

Description: □ This sets the rotor angles in 'theRotations' objects. This is equivalent to
SolidSys::theRotations.setRotorAngles(wr, beta, chi)

Example □ SolidSys moo(3);

Usage: moo.setRotorAngles(wr*0.1*PI2, 54.7*DEG2RAD);
//this is the same as above
moo.theRotations.setRotorAngles(wr*0.1*PI2, 54.7*DEG2RAD);

SolidSys::other

Function: **void setSpinMats();**

Input: void

Description: This sets all the Spin Tensors for each interaction in the SolidSys. If you add an interaction via 'mySys.csa.push_back(...)' then you must call these two functions afterwards to get correct hamiltonans 'setSpinMats()' and 'setCrystalAs().' Using the functions 'addCsa', etc. will automatically call these functions, however, if you plan to add many interactions, the 'addCsa', etc functions are much slower. It is better to use the 'push_back' functions for each Interaction list and THEN call the two function 'setCrystalAs()' and 'setSpinMats()'

Example SolidSys A(3);

Usage: A.csa.push_back(Csa(400, 0, 0, 1));

A.setSpinMats();

A.setCrystalAs();

SolidSys::other

Function:

□ **int size();**

Input:

□ void

Description:

□ Returns the number of Spins in the system.

Example Usage:

□ `SolidSys moo(3);`
`moo.size(); //returns 3`

--Public Vars

--- powder::{type}

--Constructors

--- powder

--- powder

--- powder

--- powder

--- powder

--Element Extraction

--- theta, phi,

gamma, weight

--IO

--- read

--Other Functions

--- calcPow

--- size

Examples

--- 1) the iterator

```
class powder { ...
```

This class holds the main functions and iterators to either 1) generate crystal angles from specific algorithms (zcw, alderman, sophie, rectangular, spherical) and 2) read crystal orientations from a file.

The interface is seamless between files and functional forms.

powder::Public Vars

Function:

- powder::zcw**
- powder::alderman**
- powder::sophie**
- powder::rect**
- powder::sphere**
- powder::liquid**
- powder::none**

Input:

-

Description:

- This is an enum inside the class to indicate what type of powder average generator you wish to use
 - powder::zcw--> one of the best for axially symmetric systems (like dipoles)
 - powder::alderman--> the alderman octant method
 - powder::sophie--> the sophie method of generations
 - powder::rect --> a rectangular grid weighted by sin(theta)
 - powder::sphere --> a spherical grid
 - powder::liquid --> ONE powder angle (typically (0,0,0))
 - powder::none--> none, reading from a file

Example

- powder myPow(powder::zcw, 616, 233);

Usage:

powder::constructor

Function: `powder(std::string method);`

Input: `method-->a string that can be "zcw", "alderman", "sophie", "rect", "sphere", "liquid", or a file name`

Description: `Will construct a powder from the known types, if 'method' is not one of the known types it will attempt to read the file of that name.`

Example `powder myPow("zcw"); //a zcw powder`

Usage: `powder myPow2("/usr/cyrstals/rep.pow"); //read a file`

powder::constructor

Function: □ **powder(pTypes method);**

Input: □ method--> a powder function method. it can be
powder::zcw, powder::alderman, powder::sophie, powder::rect, powder::sphere, powder::liquid,
powder::none

Description: □ Sets up a powder object using the 'powder::{type}' enum.

Example □ powder myPow(powder::zcw); //the zcw algorithm
Usage:

powder::constructor

Function: **powder(std::string method, int tstep, int pstep, int gammast=1);**

Input: method-->a string that can be "zcw", "alderman", "sophie", "rect", "sphere", "liquid", or a file name
tstep--> number of 'theta' steps
pstep--> number of 'phi' steps
gammast--> number of 'gamma' steps

Description: Constructs a powder object given with the algorithm 'method' with the parameters 'tstep' for the number of theta steps, 'pstep' for the number of phi steps, and 'gammast' for the number of gamma steps. If method is a file name the 'tstep', 'pstep', and 'gammast' are ignored

Example powder myPow("zcw" , 616 , 233);
Usage:

powder::constructor

Function: **powder(pTypes method, int tstep, int pstep, int gammast=1);**

Input: method--> a pTypes value that can be

powder::zcw, powder::alderman, powder::sophie, powder::rect, powder::sphere, powder::liquid,
powder::none

tstep--> number of 'theta' steps

pstep--> number of 'phi' steps

gammast--> number of 'gamma' steps

Description: Constructs a powder object given with the algorithm 'method' with the parameters 'tstep' for the number of theta steps, 'pstep' for the number of phi steps, and 'gammast' for the number of gamma steps.

Example powder(powder::alderman, 20, 20);
Usage:

powder::constructor

Function:

- **powder(double oneth, double oneph, double oneg=0.0);**

Input:

- oneth--> a theta value in RADIANS
- oneph--> a phi value in RADIANS
- oneg--> a gamma value in RADIANS

Description:

- Constructs a SINGLE crystal point given by the input angles.

Example Usage:

- `powder myPow(pi/4.0, pi/5.0);`

powder::element extraction

Function:

```
□ double theta(int i)
  double phi(int i);
  double gamma(int i);
  double weight(int i);

  double theta(); //FOR ITERATORS
  double phi(); //FOR ITERATORS
  double gamma(); //FOR ITERATORS
  double weight(); //FOR ITERATORS
```

Input:

- i--> the index of the powder point

Description:

- Returns the theta, phi, gamma, or the weight of the crystal point, 'i.'

Example Usage:

- powder myPow("zdw" , 616 , 23);
 myPow.theta(4);

powder::IO

Function: **void read(std::string fname);**
 void read(const char *fname);
 void read(std::ifstream &infile);

Input: fname--> a file name
 infile--> an input stream

Description: Reads a file for the powder angles. It will ignore lines that begin with '#' or '%'. The file can must be set up like so

If only a two column file then the file should be
{phi} {theta}

If only a three column file then the file should be
{phi} {theta} {weight}

If only a four column file then the file should be
{phi} {theta} {gamma} {weight}

ALL angles should be in RADIANS

Example **powder myPow;**
Usage: **myPow.read("/usr/cyrsts/mycrsy.pow");**

powder::other

Function:

- `void calcPow();`
- `void calcPow(std::string method, int tstep, int pstep, int gammast=1);`
- `void calcPow(pTypes method, int tstep, int pstep, int gammast=1);`

Input:

- `method`--> if a string can be "zcw", "alderman", "sophie", "rect", "sphere", "liquid", or a file name. If it is a pTypes it can be powder::zcw, powder::alderman, powder::sophie, powder::rect, powder::sphere, powder::liquid, powder::none

`tstep`--> the theta step parameter

`pstep`--> the phi step parameter

`gammast`--> the gamma step parameter

Description:

- Either calculates the powder average lists from previously entered data, OR from the input methods (as a string or pTypes).

Example

- `powder myPow;`

Usage:

- `myPow.calcPow(powder::zec, 616, 233);`

- `myPow.calcPow("zcw", 616, 233); //same as above`

powder::other

Function:

int size()

Input:

void

Description:

Return s how many powder points are in the object.

Example Usage:

`powder myPow("zCW" , 616 , 233);`
`cout<<myPow.size(); //prints our '616'`

Using the powder iterator

This example shows you how to use the powder iterator.

```
//declare a powder
powder myPow(powder::zcw, 616, 233);
//declare the iterator
powder::iterator iter(myPow);

//declare a solid sys
SolidSys A(1);
//add an interaction
A.addCsa(Csa(4000, 2000, 0, 1));

//loop though the iterator dumping out the
// Hamiltonian at each powder point
double t1=0.0, t2=1.0;
while(iter)
{
    A.setPowderAngles(iter.theta(), iter.phi(), iter.gamma());
    cout<<A.Hamiltonian(t1,t2)<<endl;
    ++iter;
}
```

--Constructors**--- compute****--- compute****--- compute****--Element Extraction****--- wr, sweepWidth****--Assignments****--- sweepWidth****--- wr, setWr****--Other Functions****--- FID**

```
template<class HamilGen_t>
class compute { ...
```

This class encompasses the algorithm called gamma-compute (M. Hohwy, H. Bildsoe, H. J. Jakobsen, and N. C. Nielsen, *J. Magn. Reson.* 136, 6 (1999)).

This algorithm greatly speeds up the calculation of FIDs for time dependant periodic Hamiltonians (i.e. spinning Hamiltonians) for equally spaced time steps. The algorithm can be roughly thought of as performing a Fourier Transform of propagator space. Thus it only works for equally spaced time steps and periodic time dependences.

It also performs an additional speed up in terms of elimination of the gamma powder angle. This powder angle can be related to various permutations on the time evolution propagators, thus eliminating the need to calculate those propagators for additional gamma angles.

It does have some constraints which are embedded in the algorithm itself. The algorithm divides a single period (usually a rotor cycle) into N segments. It calculates a propagator for each of the time segments. The number of segments is directly related to the sweep width of the simulation. In order for the algorithm to work the spinning speed must be some integer divisor of the sweep width and the sweep width must be larger than the spinning speed. So if I spin at 5 kHz then the only allowed sweep widths would be 5 kHz, 10 kHz, 15 kHz, etc. The larger the sweep width, as compared to the spinning speed the longer the algorithm takes to calculate the FID because there are more segments.

This class will automatically adjust the sweep width in accordance to the entered spinning speed and sweep width. After you use the algorithm to calculate an FID, you should use the sweep width and time steps from this class to get the accurate frequency and time of each point.

!!NOTE!!

It maintains a pointer to a 'HamilGen_t' class that MUST HAVE the function
matrix Hamiltonian(double t1, double t2, double wr);

where t1 is the beginning time for a given time step, t2 is the end time for a given time step, and wr is the rotor speed. The 'SolidSys' class or any other derived class will have this function. You can (and are encouraged) to write your own Hamiltonian that overrides the SolidSys classes function. 'compute' maintains a pointer to the previously declared HamilGen_t class, thus IF the HamilGen_t object is destroyed before the compute object, the compute object will FAIL to work and probably crash your machine.

If you need to use the powder angles, then make sure you perform a

'setPowderAngles' BEFORE you use compute. Also to set the Rotor angle, you MUST call 'setRotorAngles' BEFORE compute is called.

This class is used by 'oneFID' to calculate the general FID. Compute is used to calculate the spinning FIDs, but 'oneFID' can calculate the FID for either static or spinning and integrate over powder angles (and can do it in parallel). So in short, this class needs to be used only if you wish to devlope another class like 'oneFID.' For most instances I would recommend you use the 'oneFID' class.

compute::constructor

Function:

- compute()**

Input:

- void

Description:

- Empty constructor. Sets the default values
spinning speed=0
sweep width=0
HamilGen_t *=NULL

Example Usage:

- SolidSys A;
compute<SolidSys> myCom;

compute::constructor

Function: **compute(HamilGen_t &in)**

Input: in--> a class that contains the function 'Hamiltonian(t1, t2,wr)'

Description: The constructor that sets the pointer to the Hamiltonian Generation class.

Example Usage: SolidSys A(2);
A.addCsa(Csa(100, 0,0,1));
compute<SolidSys> myC(A); //sets the ptr to A

compute::constructor

Function: `compute(HamilGen_t &in, double wr,double sw, double tminin, double tmaxin)`

Input: in--> a class to generate the Hamiltonina via the function 'Hamiltonian(double t1, double t2, double wr).'
wr--> the rotor speed in Hz
sw--> the desired sweep width in Hz
tminin--> the start time for the rotor cycle
tmaxin--> the end time, typically tminin+1.0/wr

Description: The basic constructor that sets the pointer to the HamilGen_t class, sets the rotor speed (wr) and the sweep width (sw). It will automatically calculate the required sweep width required for the input spinning speed. The time 'tminin' is the start time for a rotor cycle and tmaxin is typically the time for ONE rotor cycle (1.0/wr) after tminin (tminin+1.0/wr).

Example `double wr=1000.0, sw=30000.0;`

Usage: `SolidSys A(1);
A.addCsa(Csa(300, 2000,0, 0));
compute<SolidSys> myC(A, wr, sw, 0.0, 1.0/wr);`

compute::element extraction

- Function:** **double wr();**
 double sweepWidth();
- Input:** void
- Description:** Returns the current values for the Rotor Speed (wr()) and the Sweep width (sweepWidth()).
- Example Usage:** SolidSys A(1);
A.addCsa(Csa(300, 4000, 0, 0));
double wr=5000.0, sw=30000;
compute<SolidSys> myC(A, wr, sw, 0.0, 1.0/wr);
cout<<myC.wr(); //prints out '5000'
cout<<myC.sweepWidth(); //prints out '30000'

compute::assignments

Function: **void sweepWidth(double in);**
 void setSweepWidth(double in);

Input: in--> the new sweep width in Hz

Description: Sets the sweep width (in Hz). It will then recalculate the necessary steps and change the sweep width to the closest value to the input.

Example SolidSys A(1);

Usage: A.addCsa(Csa(300, 4000, 0, 0));
double wr=5000.0, sw=30000;
compute<SolidSys> myC(A, wr, sw, 0.0, 1.0/wr);
myC.sweepWidth(35000.0); //change the sweepwidth

compute::assignments

Function: **void wr(double in);**
void setWr(double in)

Input: in--> the new spinning speed in Hz

Description: Sets the spinnig speed (in Hz). IT will then recalculate the nessesary steps and new sweep width if they need to change.

Example SolidSys A(1);

Usage: A.addCsa(Csa(300, 4000, 0, 0));
double wr=5000.0, sw=30000;
compute<SolidSys> myC(A, wr, sw, 0.0, 1.0/wr);
myC.wr(6000.0); //change the rotor speed to 6 kHz

compute::other

Function: `□ Vector<complex> FID(matrix_T &ro, matrix &det, int npts);`

Input: `□ ro-->` an initial density matrix
`det-->` the detect operator
`npts-->` the number of points in the FID

Description: `□` Computes the FID given an input initial density matrix (of any matrix type), a detection matrix and the number of points in the FID requested.

Example `□ SolidSys A(1);`

Usage:

```
A.addCsa(Csa(300, 4000, 0, 0));
double wr=5000.0, sw=30000;
compute<SolidSys> myC(A, wr, sw, 0.0, 1.0/wr);
int npts=512;
//must set the initial angles
A.setPowderAngles(0.0,0.0,0.0);
double magA=acos(1.0/sqrt(3.0));

//must set the rotor angle
A.setRotorAngles(0.0, magA);

//the typcall ideal 90 degree pulse
smatrix roeq=A.Ix();
//the standard detection matrix
matrix detect=A.Ip();

//calculate the FID
Vector<complex> fid=myC.FID(roeq, detect, npts);
```

--Public Vars

--- Controller

--Constructors

--- oneFID

--- oneFID

--- oneFID

--- oneFID

--Element Extraction

--- npts, size

--- startTime

--- sw

--- wr

--Assignments

--- setNpts

--- setStartTime

--- setSweepWidth

--- setWr

--Other Functions

--- FID

--- FID

template<class Func_T>

class oneFID :

public Func_T

{...

This class contains all the optimized algorithms for collection of FIDs in most any NMR situation (typically spinning, static, with powder angles, without powder angles). It will collect powder averaged FIDs in parallel. This wraps the 'compute' algorithm and the static algorithms into one master class.

In order for the class to function properly, the class 'Func_T' typically is the SolidSys class, where these functions are public to Func_T

```
matrix Hamiltonian(t1, t2, wr);
matrix Hamiltonian(wrPhase, rotorBeta, theta, phi, t1, t2);
void setPowderAngles(theta, phi, gamma);
void setRotorAngles(wrPhase, rotorBeta);
```

Any class derived from SolidSys will be a valid candidate, as typically all you may need to do is overwrite the Hamiltonian functions with new ones.

If either the rotor speed OR or the rotor angle is zero, then the static algorithm is used.

!!NOTE!!

If you use a derived class that has 'Pulses' (like decoupling) during the FID collection phase, usage of the 'compute' algorithm will be invalid UNLESS the pulse power (amplitude) is a multiple of the rotor speed. Even the static (non-spinning) case FID will be incorrect as the Hamiltonian is NOT time independent. There are few ways to treat this case in any optimal sense other than brute-force (performing a Dyson Time series explicitly). You could treat a continuous radiation (periodic pulse Hamiltonian) similarly to the spinning rotor simulation. However, the 'wr' would be the Pulse amplitude and you would have to write the valid Hamiltonian for these cases (the SolidSys Hamiltonians assume the 'wr' is for the rotor spinning speed).

oneFID::Public Vars

Function: **MPIcontroller Controller**

Input:

Description: This is the oneFID's MPI controller. If it is not set, then none of the algorithms will be calculated in parallel. If it is set, then the oneFID function that rely on a 'powder' object will be calculated in parallel.

Example

Usage: MPIworld.start(argc, argv); //start MPI

```
SolidSys A(3);
A.addCsa(Csa(300, 3400, 0, 0));
//a fid with 512 pts,
// a sweep width of 30 kHz, spinning at
// 3 kHz at the magic angle
oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG2RAD);
//set the 'SolidSys' class to A
myFID.setFunction(A);
//set the objects controller
myFID.Controller=MPIworld;

powder mypow(powder::zcw, 233, 144);
matrix roeq=A.Ix();
matrix detect=A.Ip();
//this will be caculated in parallel
Vector<complex> fid=myFID.FID(mypow, roeq, detect);
```

oneFID::constructor

Function:

- `template<class Func_T>`
- `oneFID<Func_T>()`

Input:

- `void`

Description:

- Null constructor that sets the defulat values

`wr=0`
`rotorAngle=0`
`npts=0`
`sweepwidth=20000`

Example Usage:

- `SolidSys A(2);`
- `oneFID<SolidSys> myFid;`

oneFID::constructor

Function: **oneFID(int npts, double sw=20000, double wr=0, double rotb=0, startT=0)**

Input:

- npts--> the number of points in the fid
- sw--> the sweep width in Hz (default=20 kHz)
- wr--> the rotor spinning speed in Hz (default=0)
- rotb--> the angle in RADIANS the rotor makes with the magnetic field (default=0)
- startT--> the starting time in seconds (default=0)

Description: Sets up a oneFID object with an empty Func_T. To set the 'Func_T' use the 'setFunction' function.

Example SolidSys A(3);

Usage: A.addCsa(Csa(300, 3400, 0, 0));

```
//a fid with 512 pts,  
// a sweep width of 30 kHz, spinning at  
// 3 kHz at the magic angle  
oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);  
//set the 'SolidSys' class to A  
myFID.setFunction(A);
```

oneFID::constructor

Function: `oneFID(Func_T &func, int npts, double sw=20000, double wr=0, double rotb=0, double startT=0)`

Input: func--> the function base to the oneFID object

npts--> the number of points in the fid

sw--> the sweep width in Hz (default=20 kHz)

wr--> the rotor spinning speed in Hz (default=0)

rotb--> the angle in RADIANS the rotor makes with the magnetic field (default=0)

startT--> the starting time in seconds (default=0)

Description: Sets up a oneFID object with the Hamiltonian function generator given by 'func.'

Example `SolidSys A(3);`

Usage: `A.addCsa(Csa(300, 3400, 0, 0));`

`//a fid with 512 pts, from the 'A' system`

`// a sweep width of 30 kHz, spinning at`

`// 3 kHz at the magic angle`

`oneFID<SolidSys> myFid(A, 512, 30000, 2000, 54.7*DEG3RAD);`

oneFID::constructor

Function: `oneFID(const oneFID &cp);`

Input: `cp-->` another oneFID object.

Description: `The copy constructor.`

Example Usage: `oneFID<SolidSys> myFID(512, 30000);
oneFID<SolidSys> myFID2(myFID); //copied`

oneFID::element extraction

Function: `int npts();`
 `int size();`

Input: `void`

Description: Returns the size of the desired FID vector.

Example Usage: `//an oneFID object`
`oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
`cout<<myFid.npts(); //prints '512'`

oneFID::element extraction

Function: **double startTime()**

Input: void

Description: Returns the start time for the fid.

Example Usage: `//an oneFID object
oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);
cout<<myFid.startTime(); //prints '0'`

oneFID::element extraction

Function:	<input type="checkbox"/> double sw(); <input type="checkbox"/> double sweepWidth();
Input:	<input type="checkbox"/> void
Description:	<input type="checkbox"/> Returns the current sweep width.
Example Usage:	<input type="checkbox"/> <code>//a oneFID object</code> <code>oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);</code> <code>cout<<myFid.sw(); //prints '30000'</code>

oneFID::element extraction

Function: **double wr();**
 double rotorSpeed();

Input: void

Description: Return the current rotor speed.

Example Usage: `//a fid object`
`oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
`cout<<myFid.wr(); //prints '2000'`

oneFID::assignments

- Function:**
- `void setNpts(int nptsNew);`
 - `void setSize(int nptsNew);`
- Input:**
- `nptsNew-->` the new size of the FID vector
- Description:**
- Sets the size of the FID vector.
- Example Usage:**
- `//an oneFID object`
 - `oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
 - `myFid.setNpts(1024); //sets the FID size to 1024`

oneFID::assignments

Function: **void setStartTime(double newStart)**

Input: newStart--> the new start time for the FID.

Description: Sets the start time of the FID.

Example Usage: `//an oneFID object`
`oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
`myFid.setStartTime(0.03); //sets start Time ro 0.03 seconds`

oneFID::assignments

- Function:**
- `void setSweepWidth(double swNew);`
 - `void setSw(double swNew);`
- Input:**
- `swNew--> the new Sweep Width in Hz`
- Description:**
- Sets the sweep width of the FID.
- Example Usage:**
- `//an oneFID object`
 - `oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
 - `myFid.setSw(60000); //sets the sweep to 60 kHz`

oneFID::assignments

Function:

- **void setWr(double wrNew);**
- **void setRotorSpeed(double wrNew);**

Input: □ wrNew--> the new rotor speed in Hz

Description: □ Sets the rotor speed (in Hz) for the FID object.

Example Usage: □ `//an oneFID object`
`oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG3RAD);`
`myFid.setWr(6000); //sets the rotor speed to 6 kHz`

oneFID::other

Function: `□ Vector<complex> FID(powder &myPows, matrix &roEQ, matrix &det);`

Input: `□ myPows--> a powder object containing the desired integration powder angles
roEQ--> an initial density matrix (must be the same hilbert space size as the Func_T class)
det--> a detection matrix (must be the same hilbert space size as the Func_T class)`

Description: `□ Collects an FID given a powder object, an intial density matrix and a detection matrix. You must have define the function class before this will work.`

Example `□ SolidSys A(3);`

Usage: `A.addCsa(Csa(300, 3400, 0, 0));
//a fid with 512 pts,
// a sweep width of 30 kHz, spinning at
// 3 kHz at the magic angle
oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG2RAD);
//set the 'SolidSys' class to A
myFID.setFunction(A);
//set the objects conbtrller
myFID.Controller=MPIworld;

powder mypow(powder::zcw, 233, 144);
matrix roeq=A.Ix();
matrix detect=A.Ip();
//this will be caculated in parallel
Vector<complex> fid=myFID.FID(mypow, roeq, detect);`

oneFID::other

Function: `Vector<complex> FID(matrix &roEQ, matrix &det, double theta=0, double phi=0, double gamma=0);`

Input: `roEQ`--> the initial density matrix

`det`--> the detection matrix

`theta`--> the 'theta' Powder angle (in RADIANS)

`phi`--> the 'phi' Powder angle (in RADIANS)

`gamma`--> the 'gamma' Powder angle (in RADIANS)

Description: `Will calculate an FID for ONE Euler angle (phi, theta, gamma).`

Example `SolidSys A(3);`

Usage: `A.addCsa(Csa(300, 3400, 0, 0));`

`//a fid with 512 pts,`

`// a sweep width of 30 kHz, spinning at`

`// 3 kHz at the magic angle`

`oneFID<SolidSys> myFid(512, 30000, 2000, 54.7*DEG2RAD);`

`//set the 'SolidSys' class to A`

`myFID.setFunction(A);`

`//set the objects controller`

`myFID.Controller=MPIworld;`

`matrix roeq=A.Ix();`

`matrix detect=A.Ip();`

`//this will be caculated for one powder angle`

`Vector<complex> fid=myFID.FID(roeq, detect, PI/3.0, PI/4.0);`