Chapter 1 - Overview

The Penn State University Digital Ostensive Magnetic Resonance Image (PSUdo MRI) simulator engine is software developed at Penn State University's Center for Nuclear Magnetic Resonance Research (PSUCNMRR). It is a versatile MRI simulator considering static, gradient, and RF field distributions, including SAR and noise correlation for multiple transmit and receive coils in 3D. PSUdoMRI was developed first to accurately predict how realistic distributions of the electromagnetic fields (DC, switched gradients, and radiofrequency) affect the MR images for arbitrary MRI pulse sequences. It is capable of considering multiple RF coils in transmission and in reception, considering correlated noise between RF channels, and calculating the SAR distribution throughout the sample for arbitrary pulse sequences. It could be a valuable aid in the design of pulse sequences to overcome or compensate for imperfections in the field distributions due to field/tissue interactions or design constraints. It is our hope that the package will find value for other applications as well.

This manual is directed at users with at least some background in software development who are interested in discovering the inner workings of the PSUdo MRI engine. We will not discuss here the programming of the PSUdo MRI Graphical User Interface (GUI), but will cover details about the engine, which was coded in C++ and designed for cross-platform execution. We have used C++'s Object Oriented Programming style to create a collection of object classes which interact to perform complex calculations and produce multiple outputs.

As stated, this engine is also part of a larger product which includes a GUI, designed in MATLAB, which can be used to call the engine with a compact, user friendly interface. The engine can also be used ad-hoc, totally separate from the user interface. In order to do this, a more intricate knowledge of the engine's procedures and inputs is necessary.

Versions of the PSUdo MRI engine

The PSUdo MRI simulator was designed to perform as many different functions as possible, and eventually during the development of the engine, the runtime memory grew to such a size that it was impractical to bundle all of these functions into one executable code. The decision was made to separate the engine into three stand-alone versions. These three versions are thus more directed and specific to one particular function of the simulator. The result is that there is one program version to perform noise calculations, one to perform Specific Absorption Rate (SAR) calculations, and one to calculate signal.

This separation by function also makes it easier for users to focus on one particular calculation if they desire. It also means that users must be aware of the function and capability of each version. In particular, it is important to remember which files are necessary for which version and which must be omitted for successful simulations to occur.

Chapter 2 - Input Files

:TODO: note in tissue type file that 1) it is .txt, and 2) labels MUST appear in the first row (unless change is made)

File Structure for Spatially-organized Files

Here we will describe the structure of the input files in text format. The engine reads files in binary format (except for the tissue property file, which is always text format). Some simple conversion tools (in the conversion tools folder) are provided to facilitate translation from text to binary format. Also, text versions of many file types are available in the folder "PSUdoMRI/Models/Head 2mm 125MHz/txt"

The sample geometry file in text format contains a brief header followed by four long columns of numbers. The 6-row header includes (on each row, in order of appearance) values for

1) The number of rows in the four-column array,
2) The minimum and maximum x-coordinates where tissue is found,
3) The minimum and maximum y-coordinates where tissue is found,
4) The minimum and maximum z-coordinates where tissue is found,
5) The x, y, and z coordinates of the intial gradient isocenter, and
6) The grid resolution (in mm) in the x-, y-, and z-directions.

Each row of the four-column array contains integer x, y, and z coordinates and an integer tissue identifier corresponding to the "ID" in the tissue property file. After converting to binary, the sample geometry file should be given the extension ".smpl".

:TODO: this is still reading .txt in the engine for tissue property file

The tissue property file is a text file with extension ".prop". The first row of the file has been left as labels to indicate the value types in columns of following rows. Each row gives properties of a different tissue with a unique integer ID. After the ID, seven properties are given for each tissue, including T1 (in ms), T2 (in ms), proton density (fractional), chemical shift (in ppm), electrical conductivity (in S/m), mass density (in kg/m^3), and the name of the tissue.

The ΔB_0 file in text format contains a brief header followed by four long columns of numbers. The 1-row header contains only the integer number of rows in the table that follows. This number must match that of the corresponding sample geometry file. The order of spatial information in the ΔB_0 file should also match that of the geometry file, row by row. Each row of the table in the ΔB_0 file has the magnitude deviation (in ppm) from a perfectly-uniform B_0 field followed by the local gradient in B_0 (in ppm/m) in the x-, y-, and z-directions.

The B_1 field file for a given transmit or receive coil in text format contains a brief header followed by two long columns of numbers. The 1-row header contains only the integer number of rows in the table that follows. This number must match that of the corresponding sample geometry file. The order of spatial information in the B_1 field file should match that of the geometry file, row by row. Each row of the table in the B_1 field file has the complex (real and imaginary parts, in tesla) magnitude of the circularly-polarized component of the B_1 field produced by the coil. Note that for a given coil, circularly-polarized components with opposite sense of rotation are pertinent for transmission and reception, and also that quadrature coils should be driven with opposite phases in calculating these two components. Which component is pertinent in transmission and in reception depends on the whether B_0 is parallel or anti-parallel with the z-axis.

The E_1 field file for a given transmit or receive coil in text format contains a brief header followed by six long columns of numbers. The 1-row header contains only the integer number of rows in the table that follows. This number must match that of the sample geometry file. The order of spatial information in the E_1 field file should match that of the geometry file, row by row. Each row of the table in the E_1 field file has the complex (real and imaginary, in V/m) magnitudes of the x-, y-, and z-components of the electrical field produced by the coil when it is driven in the same way as to produce the corresponding pertinent B_1 field information.

:TODO: The format of the Gradient field files not yet finalized.

File Structure for Sequence Files

The Sequence file consists of a header followed by a number of different blocks, each of which can be structured very differently. The header information includes the following:

- the integer number of time steps (Nsteps) in the entire sequence,
- the number of transmit coils (Ncoils), and
- the number of receive coils.

In order of appearance, the blocks (5+Ncoils in number) describe

1) the time step sizes (in ms),
2) the receiver/kmap/ideal crushing events (all in one block with indicators 1, 0, -1, -2),
3) the real, imaginary, and frequency offset at each time for each transmit coil drive (number of blocks equal to Ncoils arranged in series, one block for each coil),
4) the x-gradient strength (in mT/m),
5) the y-gradient strength (in mT/m), and
6) the z-gradient strength (in mT/m).

Once it is past the header information, the engine reads the sequence file data assuming it is always in a loop within a block until the number Nsteps is reached, at which time it assumes it is entering the next block. Thus each block must begin with structural information for the first loop in that block. Loop structure is indicated with 2 integers: a number of elements per loop followed by a number of loops. For example, in a gradient block, a gradient strength of 12 mT/m for one timestep followed by 4 timesteps at 0 mT/m, all looped 3 times, could be represented by any of the following:

5 3 12 0 0 0 0

or

1 1 12 1 4 0 1 1 12 1 4 0 1 1 12 1 4 0

or

15 1 12 0 0 0 0 12 0 0 0 0 12 0 0 0 0

where values for number of elements in a loop are shown in red and those for number of loops are shown in blue. Thus sequence file size can be reduced to some degree with strategic use of loops. Of course it is always possible to forego loops altogether by beginning each block with Nsteps followed by 1, as shown in the third example above.

The data in the timestep block will contain a single value in ms for the length of the corresponding timestep. All field values are constant during a timestep, so timesteps will necessarily be

much smaller during definition of a shaped RF pulse than during a rectangular phase encoding gradient or a recovery period.

The receiver/kmap/crush block will contain a set of "flag" numbers indicating the receive/kmap/crush function during the corresponding timestep. The "flag" numbers in this file indicate the following functions:

Flag:	Indicates function:
1	Data is collected
0	No data is collected
-1	Sequence returns to the center of k-space, such as at the end of a TR (for use of the k-map files)
-2	Sequence returns to the center of k-space AND simulate ideal crushing, setting transverse magnetization to zero.*

*A current known issue with the simulator is that gradient and RF crushing can result in some aliasing-like banding in the signal, likely due to the discrete nature of the models.

For each timestep in each of the Ncoils blocks describing the transmit RF waveforms, three values must be given: Real and Imaginary parts of the amplitude (given as a multiplier for the information in the corresponding B1 field file) and frequency offset (in Hz).

For each timestep in each gradient waveform block is the desired gradient strength (in mT/m).

Chapter 3 - Command line

While the PSUdo MRI engine was developed concurrently with the Matlab GUI, and can be called directly from that interface, the engine can also be used directly. In order to pass the necessary parameters into the engine, however, we must use command line input, such as a DOS command or a batch file (.bat). This is how the engine receives information like B0 field strength, voxel dimensions, file names, and other data. This chapter describes the format in which these parameters must be written and gives a complete list of possible command line parameters. There will also be some discussion about how the engine parses the command line to use the parameters for a simulation, and lastly, there will also be some examples of valid command line strings.

The program call used to run the PSUdo MRI engine will start with the executable name. This will be the name of the .exe file, which can be anything selected by a user. In our case, the .exe files are called "NutateNoise.exe", "NutateSAR.exe", and "NutateSignal.exe". The call will be followed by the parameter flags, which will be listed below. As an example, let us set the region of interest parameters in the x-direction. These flags are `xMin` and `xMax`. In order to set these values, our program call would look like this:

```
NutateNoise xMin=-93 xMax=107
```

In this example, we are calling the noise calculation version of the engine and setting the x-direction values of the region of interest to include all voxels with x values in the closed interval [-93, 107] (:TODO: check inclusive/exclusive). This shows the general format of parameter input. Each parameter will have a flag, such as `xMax`, followed by the "=" sign, followed by the parameter. It is important that there are not spaces on either side of the equals sign. The flag must be followed immediately by the "=" sign, which must be followed immediately by the parameter. These options will all be separated by a space. These rules also apply to file names. They should not be surrounded by quotes, and they should also include the file's extension. For example, to use a geometry file entitled "head_model.smpl" and set the region of interest parameters as in the previous example, the command line should read:

```
NutateNoise xMin=-93 xMax=107 GeometryFile=head_model.smpl
```

There is one more type of input that must be described here. The signal calculation version of the PSUdo MRI engine takes, as a command line parameter, the B0 field strength, which can be a floating point number. When including this in the command line input, the field strength can be written with or without the decimal (i.e. `B0=3` `B0=3.0` OR `B0=3.825`).

This is the general format for command line input. The table below shows all of the possible input arguments, their function, and the engine versions that they can be used with. (:TODO: gradX, gradY, gradZ - removed?)

		Applicable engine version		
Parameter	Description	Noise	SAR	Signal
`xMin`	Minimum x-value for region of interest	x	x	x
`xMax`	Maximum x-value for region of interest	x	x	x
`yMin`	Minimum y-value for region of interest	x	x	x
`yMax`	Maximum y-value for region of interest	x	x	x
`zMin`	Minimum z-value for region of interest	x	x	x
`zMax`	Maximum z-value for region of interest	x	x	x
`TissueTypeFile`	*File name of the tissue type file	x	x	x
`GeometryFile`	*File name of the geometry file	x	x	x
`SequenceFile`	*File name of the sequence file	x	x	x
`xWid`	Voxel width in the x-direction	x		x
`yWid`	Voxel width in the y-direction	x		x
`zWid`	Voxel width in the z-direction	x		x
`Thread`	Number of threads to use for parallel processing		x	x
`B1PlusFile`	File name of the B1+ file		x	x
`E1MinsFile`	File name of the E1- file	x		
`KNoiseFile`	File name to be used for the k-noise *output* file	x		
`E1PlusFile`	File name of the E1+ file		x	
`SARFile`	File name to be used for the SAR *output* file		x	
`B0`	B0 field strength			x
`xCtr`	Center of x gradient			x
`yCtr`	Center of y gradient			x
`zCtr`	Center of z gradient			x
`DelB0File`	File name of delta B0 file			x
`B1MinsFile`	File name of B1- file			x
`KSpaceFile`	File name to be used for k-space *output* file			x
`KMapFile`	File name to be used for k-map *output* file			x

Table 1 - * indicates that this is a required command line input

(:TODO: verify which are required)

Some of the values passed in via the command line are necessary for execution of the engine. These are indicated above and if any one of those is not provided at the program call, execution will terminate, asking for the user to correct errors and restart. All input flags not indicated as required command line input can be omitted without causing the program to terminate without simulating.

Some of the non-required inputs can also be used to edit simulation parameters from their original, default values. For example, the values for `xWid`, `yWid`, `zWid`, `xCtr`, `yCtr`, and `zCtr` are all provided in the geometry file header. If they are not included in the command line, the engine will default to parameters found in the geometry file header. The PSUdo MRI engine gives users the option to use the command line input as an easy way to modify execution parameters without editing the original file information. For example, the user can scale the model in any direction and also shift the center-of-gradient position using the command line values. In summary, command line input is always used when provided, but the engine will use file header values when command line input is not present.

Parsing

The command line is processed at the start of all three versions of the PSUdo MRI engine. This is always done using an instance of the `CmdInterp` class. This class is designed as a buffer to hold the command line values before they are placed in their appropriate locations before simulation. `CmdInterp` is different for each version of the engine because each version receives different command line parameters,

however the command line parsing follows the same principle through all 3 versions. The `CmdInterp` class has a buffer for each possible input data along with a flag variable that indicates whether or not that variable has been processed by the command line interpreter. When the default constructor of `CmdInterp` is called, all buffers are cleared and all of these flags are set to false. Upon completion of the command line, we can then check to ensure all of the buffer flags have been set to true to verify that all of the necessary data has been provided.

In order to parse the command line, which is done in the `interpCmd` member function, the engine will look at each "prase" of the command line delimited by spaces. It will then search that "phrase" for the equals sign (=), at which point we can separate the command flag from the data it represents. In the function, these two are represented by the `option` and `parameter` string variables. One "phrase" delimited by spaces is copied into the `option` variable and from here, we search the string for the equals sign and place a pointer, `ptr`, there. We can then write everything after the pointer in `option` and place it in `parameter`, then delete it from `option`. At this point, we should have only the command flag in `option` and the variable value in `parameter`.

At this point, the variable, `cmdMap`, will come into play. This is a map object, which links one type of data to another. In this case, it links strings to an enumeration of command options - an easily searchable, user defined list of possibilities. These two things, hand-in-hand, make it possible to use a `switch` statement that reads easily to the programmer. We link the input string to our self-defined list of options, then search for the option linked to the input string found in `option`. The `switch` statement will find the matching `case` statement, and the data we stored in `parameter` can be placed in the appropriate data buffer until it can then be copied to its final location.

The system we use to read the command line supports a range of user options, default overrides, and flexibility with the PSUdo MRI engine. The system was developed to produce as few runtime errors as possible. Keep in mind that with this system, a user may accidentally enter the same command flag twice. If this happens, (as long as it is not a type that accepts multiple inputs) the second input will overwrite the first. This is a possible source of undetected errors in the command line and something for users to keep in mind when debugging erroneous output.

Multiple files of the same type - *Input files*

In many cases while using the PSUdo MRI engine, a user will want to simulate multiple transmit and/or multiple receive coils. In order to do this, they must provide the file names for several files which will be associated with the same command flag. For example, in the signal calculation version of the engine, one might be simulating using multiple transmit coils. In order to do this, the engine will need to be provided with a B1+ file for each coil. The method for providing multiple file names of the same type is simply to re-use the command flag associated with that type. The B1+ files, for example, would simply be included as follows.

```
... B1PlusFile=model1_B1p_1.bin B1PlusFile= model1_B1p_2.bin B1PlusFile= model1_B1p_3.bin
B1PlusFile= model1_B1p_4.bin ...
```

Notice that the command flag does not change in any way. It is simply re-stated in the command line. The only thing changing in the example is the file name. The engine will input the files in the order they appear in the command line. This order is important because some output files depend on the receive coil number. To later determine which coil produced a particular file, users should make note of the order of input files or use file names with reference numbers for input files. (:TODO: ask about importance of order - does it affect simulation? are there different file types about the same coil?)

It is also possible for a user to use the same file for several coils. Files are read sequentially, meaning one is closed before the other is opened. If the same file is used many times, this will not cause an error. The command line should hold the same command flag with the same file names, repeated for as many coils you wish to simulate.

Multiple files of the same type - *Output files*

The PSUdo MRI engine will also generally produce more than one output file of the same type. For output files, however, the user need only to provide one file name. This file name forms the base of the output file names. Each file will have the same name and will be appended with the number of the producing coil. For example, if the user enters `kSpaceFile=kSpace.bin` in the command line, the output file names will be "kSpace1.bin", "kSpace2.bin", "kSpace3.bin", and so on. For k-space files, the number appended indicates the receive coil that produced that k-space. The receive coil number is a function of the order in which input files appeared in the command line. This is why, generally, input file names should include a reference number, and should appear in the command line in numbered order.

PSUdo MRI does not support multiple output file names. One name should be provided, and this file name will be appended with coil names. If multiple output file names are provided, only the one that appears last in the command line will be used and appended with coil numbers.

Sample input strings

Now that we have discussed the command line details, we will provide some examples of calls to the PSUdo MRI engine which do not generate command line errors (given that the provided files exist).

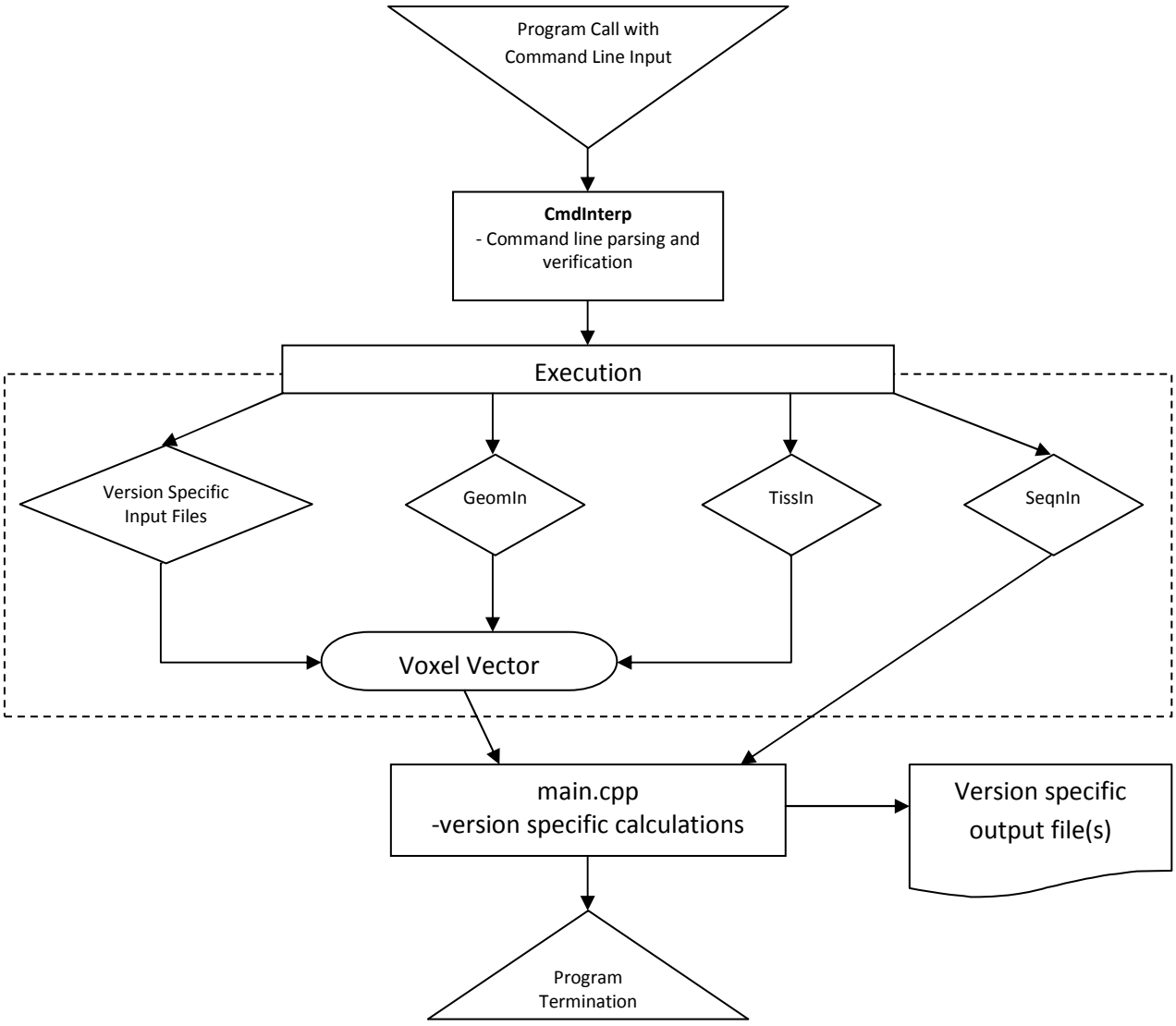
```
NutateNoise xMin=-93 xMax=107 yMin=-84 yMax=80 GeometryFile=head_model.smpl
TissueFile=tissType.txt SequenceFile=sqn512.bin kNoiseFile=NoiseOut.bin(<--:TODO: .bin?--
>)E1MinsFile=E1M_coil1.bin E1MinsFile=E1M_coil2.bin E1MinsFile=E1M_coil3.bin
E1MinsFile=E1M_coil4.bin
```

The above command line will call the noise calculation version of the PSUdo MRI engine. We have set the region of interest for the x- and y-directions, while the z-direction is not limited. The voxels to be used for this run of the engine will fall in the x-range [-93, 107], the y-range [-84, 80], and will include all voxels in the z-direction which meet these x- and y-direction constraints. Notice also that we have omitted the `xWid`, `yWid`, and `zWid` parameters. Therefore, these will be set to the values found in the geometry file header by default. We have also included, in this program call, file names for the geometry file, tissue type file, sequence file, k-noise output file, and four E1-minus files. Notice also that there are no quotes around filenames, no spaces around equals signs (=), and that multiple E1-minus files were provided using the same command flag for each.

Chapter 4 - Anatomy of a run

The PSUdo MRI engine's three versions each produce a different output and have a different set of operations they perform during their execution. There are some similarities between the three program versions. All three begin their execution in much the same way - command line processing and file reading. This chapter provides a walkthrough, using the C++ code, of what happens during the execution of each version of the PSUdo MRI engine.

All three versions startup similarly. The first lines in the function, `int main(int argc, char *argv[])` define an instance of the class `CmdInterp`. This calls the default constructor of the class which will initialize all of the class's variables and flags. Following this, there is a call to `cmdInterp.interpCmd(argc, argv)`, which will parse the command line, locating each input flag and placing the command line data in the appropriate buffer location. Upon successful completion of `interpCmd`, each of the command line variables has been read and stored, and the associated flags in `cmdInterp` have been set. Once this is done, `cmdInterp` has performed is essential functions. The class will then be referenced throughout the program for command line values such as filenames. Next, we define an instance of the `Execution` class. The default constructor will in turn define an instance of the `GeomIn`, `TissIn`, and `SeqnIn` classes, calling their default constructors. These constructors call all of the necessary routines to read file information from the geometry, tissue, and sequence files, respectively. Each version of PSUdo MRI will also include other input files specific to that version's function, but the three mentioned here are used for all versions. Other input files will also be input using a constructor for a class defined specifically for that file. This also will happen in the `Execution` constructor and, once all files have been read, the constructor calls routines to link all of the necessary data together. This involves calling member functions on the `phantom` vector in order to set each `voxel`'s parameters. Once the `Execution` constructor is complete, we can then use the instance of that class, `run`, to access all of the data read from the input files. The figure below depicts the flow of execution of the engine. Following the figure are version specific descriptions of run-time processes.



As the figure shows, each version of the PSUdo MRI engine takes a different path after the `Execution` class has performed its functions. It is also important to note that the `Execution` class is slightly different for each build and each contains some version-specific input files and routines. From here, we will discuss each version of the PSUdo MRI engine separately.

Noise Calculation

The noise calculation version of PSUdo MRI runs through the signal acquisition data from the sequence file searching for acquisition windows. Whenever an acquisition time is found, an n x n matrix is filled by random floating point numbers having a Gaussian distribution on the interval (-1,1) (:TODO: source). This is done with the function `rdmMat(info.numRx)`, where `info.numRx`, the number of receive coils, becomes the matrix size, n. This random matrix is called `zMat`. We then multiply each entry of `zMat` by the corresponding entry in `aMat`, (:TODO: describe aMat), and sum each row of the resulting matrix. Finally, these row sums are multiplied by $\sqrt{\frac{1}{acq\ time}}$ and stored in the `kNoiseData` matrix, where *acq time* is the length of the acquisition window as defined by the sequence file. This process will generate one

noise value per receive coil per acquisition window. From this, we generate a matrix that can be indexed by receive coil and acquisition window number.

This matrix is used to output one file per coil of K-noise data (:TODO: verify name). The filenames will be modified to show coil number.

SAR Calculation

(:TODO: discuss SAR calc specific routines called in Execution constructor.)

After initializing all data and reading in the necessary files, the SAR calculation version of the PSUdo MRI simulator begins looping through each time index of the sequence file. For each time index, the vector `tempRF` is filled with the values from each transmit coil at that particular time. We use this information to then calculate the SAR for every voxel in the simulation. This is done by calling the `calSAR` routine, which is a public member function of the `Voxel` class. (:TODO:? describe calSAR?) The result of `calSAR` is added to `sarMapData`, which accumulates the SAR due to each time step in the sequence for every voxel in the simulation. Once the engine has run through all of the time steps in the sequence file, the `sarMapData` vector will have accumulated the total SAR for each voxel over the entire sequence, and this data will be written to a file with the name provided in the command line.

Signal Calculation

(:TODO: describe routines in Execution constructor)

After the initialization and file read-in, the signal calculation version of the PSUdo MRI engine will loop through each time index of the sequence file. For each time index, the vectors `tempRF` and `tempDelFreq` are filled with the values from each transmit coil at that particular time. We also set the `tempGrad` variable with the current gradient values. At this point, the engine will check to see whether or not the current time step is designated as an acquisition block.

During an acquisition time step, we start by calculating the effective B-field, using `calBeff`, for every voxel in the sample geometry and applying a rotation to it, using `appRot`. These calculations take into account all of the present coil values and also the length of the time step. It is important to note, though, that the rotation that happens at this point simulates only *half* of the total time step. (:TODO: make note about order if main.cpp does not change) Next, the engine cycles through receive coils. For each coil, we sum the result of the `acqSignal` function over all of the voxels. This information is placed at the end of the `kSpaceData` vector, where the first dimension refers to receive coil number and the second is that coil's k-space data. At this point the signal has been acquired and we complete the rotation for each voxel. This is again done using `appRot` and an argument of *timestep/2*.

Also during acquisition time steps, we must generate the k-map data (:TODO: name?). This is done in much the same fashion as signal acquisition. We calculate the k-map data (:TODO: name?) using half the time step length, store the result, and repeat the calculations, again using half the time step length.

All of the above will happen if the engine finds that the current time step is designated as a signal acquisition. In the event that we find a non-acquisition flag in the sequence information, we calculate B-effective with `calBeff` and complete the entire rotation, using `appRot`, on each of the voxels at once, rather than in two halves. :TODO: describe what 0, -1, and -2 mean in the sequence file.

After running through all of the time steps in the sequence file, the engine will output the k-space and k-map files. There will be one k-map file and as many k-space files as there are receive coils in the simulation. Each of these k-space file names will be modified to indicate the receive coil number which produced that k-space.

Chapter 5 - Output Files

PSUdoMRI outputs four types of files: signal (extension ".ksig"), noise (extension ".nois"), k-space map, or k-map (extension ".kmap"), and SAR (extension ".sar") files. Each is structured as a single array of binary "float" numbers, and each contains different information in a specific order.

The signal file contains the real part followed by the imaginary part of the signal received in the given coil at each acquisition time point in the order of occurrence during the sequence.

The noise file contains the real part followed by the imaginary part of the noise received in the given coil at each acquisition time point in the order of occurrence during the sequence.

The k-map file contains the location in k-space (in m^{-1}) in all three directions (k_x, then k_y, then k_z) at each acquisition time point in the order of occurrence during the sequence.

The SAR file contains the integer grid coordinates (x, then y, then z) of a voxel followed by the corresponding tissue ID and SAR (in W/kg) for all voxels.