Code Documentation

Fajans Lab and & Friends

July 27, 2014

Preface

This file is intended to act as a manual for all of the code used for the Lorentz Invariance investigations. It is a work in progress and you should feel free to edit it. The version tracking is performed by git and you may download this software from Github. The Github repository is private, so send Zak an email at uphgreat@gmail.com containing your Github username if you would like access.

Contents

Chapter 1

Analysis Class

The analysis class is designed to allow the user to perform all the high level tasks necessary for the analysis, so if you want to use the code but don't want to read this whole lengthy manual, just read through this chapter. Each analysis instance is designed to work with one generate_event_times() function (since we expect to pick one and stick with it in the end). This class inherits from Matlab's handle class, so references to it act like pointers.

1.1 Set-up/Initialization

The first step to create an Analysis instance is to decide which generate_event_times() function you would like to use. Several versions are included in the code, each one residing in its own subdirectory of SimulationData/DataSets/. The names of the subdirectories are used to identify the different versions of that function. You may add your own version in its own subdirectory there if you desire. Make sure no functions with the name generate_event_times() are available in your path; that will disrupt the inner workings of the Analysis class and cause it to throw an error. The Analysis class will automatically add the appropriate directories to the path variable when an instance is created.

Now you are ready to initialize an instance of this class. To do this, change to the directory[1] SimulationData/code/Classes. Now call the constructor with the name of the generate_event_times.m folder as the only argument (just the name of the folder, not the path to it) as shown below. This will automatically create the subdirectories necessary and will warn you if there are issues.

Since the whole analysis process can take a long time, a lot of results are saved to the hard drive so that future instances of the Analysis class can have the results without having to recalculate them. Therefore, if you've already run an analysis, you can get a lot of the data back just be creating another Analysis instance for the same folder and using its load functions (which are automatically called when you create the instance for the important data).

```
>> four_month=Analysis('FourMonth')
```

[1]The paths used internally are relative to SimulationData/code/Classes, so make sure this is your working directory; do not just add it to your path

```
four_month =

  Analysis with properties:

       GENERATOR_NAME: 'FourMonth'
    signal_group_list: {1x6 cell}
            n_workers: []
        data_set_root: '../../DataSets/FourMonth'
```

1.2 Running the Analysis

Running the analysis is a two-step process. First, tell the the analysis instance what kind of signals you would like it to simulate. The easiest way to do this is to use Analysis.simple_add_sine(). This function takes at least one argument: the amplitude of the fractional charge variation, which should be on the order of $\sim 1 \times 10^{-8}$. You may optionally also specify 'period', 'phase', or 'n_sets'.

You can also add your own function for picking charge values as a function of time, velocity, etc. Be careful when doing this though, because Matlab stores all workspace variables when you create a function handle, which can include all the tracer data if you have an Analysis instance open. This causes issues when the code saves the function handle to the hard drive because it will include a copy of all the tracer data. To add your own function, first add the desired signal_group instances to the analysis instance by using the analysis.add_signal_group() method. The method requires a function handle as an input, and you can optionally also pass it a string, which sets the signal_group's signal_name attribute, and/or a number, which tells it how many data sets to generate. The function handle should be a function which takes a data_set and modifies it to add a signal to the raw data. Two such functions are included: signal_null() and signal_sine(). By default, two signal_groups are added when you create an analysis instance, one for each of these. You may also want to add other signal_sine groups with different amplitudes, frequencies, or phases. You may remove a signal_group by using the analysis.delete_signal_group() method, which takes a signal_name, or by using analysis.pop_signal_group() which deletes the most recently added signal_group.

Once the desired signal groups are added, the data for all the various signal groups can be generated simply by calling analysis.run(). This will create all the data_sets, use Andy's Charman IV algorithm on them, store the results, and generate a nice histogram showing the results of the first and last signal_groups.

After running the analysis, more signal_groups can be added or deleted, then analysis.run() can be called again to generate the data and the histogram. If all the desired data has been generated, you may call analysis.generate_Charman_histograms() directly and specify which signal_groups to plot and how many bins to use.

```
>> four_month.add_signal_group(@(data_set)signal_sine(data_set,0.01,'day',pi/8),'delayed
>> four_month.run()
```

```
Generating data for signal group null...
Done. Took 4.56 seconds

Generating data for signal group daily sine...
Done. Took 5.45 seconds

Generating data for signal group delayed daily sine...
Done. Took 5.00 seconds


Generating Charman histograms...
Finished generating Charman histograms
Generating Charman histograms took 0.22 seconds


>> four_month.generate_Charman_histograms({'null','daily sine','delayed daily sine'},25)
Generating Charman histograms...
Done.  Took 0.22 seconds
```

1.3 Analysis.add_signal_group()

This method creates a signal_group and adds it to analysis.signal_group_list. You must give it a function handle that accepts a data_set as an input, then returns an array of charges. You may create your own signal functions, or you may use the provided ones, which are discussed in Chapter 6.

You may also specify a signal_name, which you will use to identify the signal_group when you interact with it (plot its data, delete it, etc.). If you do not specify a name, one will be generated automatically using the Analysis.func_to_signal_name() method. Furthermore, you may specify a number of data_sets to generate for that signal_group. If you do not provide a value, it will allow the Signal_Group constructor to choose the default value (which is currently set to 100).

If you choose to use the signal_sine function for a signal_group (and you most likely will), you will have to provide some arguments in the function handle for the amplitude, period, and phase. This is because the function handle given to Analysis.add_signal_group() must take only one argument: the data_set. See the examples of valid uses below.

Another point worth noting is that the code is smart enough to stop you from adding two signal_groups with the same signal_names, and it is smart enough to stop you from adding the same function handle twice. If you attempt to do this, it will not create a new signal_group and it will display a warning.

To remove a signal_group from analysis.signal_group_list (this will also delete all its data from the hard drive), you may call Analysis.delete_signal_group() with the signal_name of the signal_group as the only argument. Another alternative is to use Analysis.pop_signal_group(), which automatically deletes the signal_group at the end of analysis.signal_group_list (which is the most recently added one).

```
>> four_month.add_signal_group(@(data_set)null(data_set),'null')
Warning: The given function name null is already in self.signal_group_list
> In Analysis>Analysis.add_signal_group at 202
>> four_month.add_signal_group(@(data_set)signal_sine(data_set,0.02,'day',0));
>> four_month.add_signal_group(@(data_set) ...
signal_sine(data_set,0.02,'day',pi/8),'delayed sine',500);
```

1.4 Analysis.generate_Charman_histograms()

This method produces a histogram that appears as a docked figure. You may call it with no arguments, in which case it will by default plot the first (which is usually the null signal_group) and the most recently added signal_group with 30 bins. Passing it a number will specify the number of bins. Passing it a cell array with signal_names will make it plot the data from the specified signal_groups.

The most bins you can reasonably have with the data from LargeSimData is ∼30, but AllSimData looks good up to at least ∼100 bins. Currently, this function does not save the figures, but they can be saved with the savefig() command.

```
>> four_month.generate_Charman_histograms({'null','daily sine','delayed daily sine'},25)
Generating Charman histograms...
Done.  Took 0.31 seconds
```

Chapter 2

Signal_Group Class

You should not need to interact directly with this class very often. Instances of this class are stored in the Analysis.signal_group_list property. The only time I expect that a person would want to interact with it directly would be if they wanted to change the number of data_sets that are generated for that signal_group. This can be done with the Signal_Group.set_n_sets() method, which takes one argument: n_sets. If n_sets is 0, the signal_group will automatically set it to its default value (right now that is 100). Note that if n_sets increases, the new data_sets will not be generated immediately. To generate additional data_sets, call analysis.run()

```
>> four_month.signal_group_list{1}.set_n_sets(1000)
>> four_month.run()
Generating data for signal group null...
Done. Took 47.21 seconds
blah blah blah....
```

Chapter 3

Data_Set Classes

This chapter discusses Data_Sets, Raw_Data_Sets and Calc_Data_Sets. The overall structure is that each instance of the Data_Set class manages an instance of the Raw_Data_Set class and an instance of the Calc_Data_Set class. Each raw_data_set contains the times and z-positions of the events, and each calc_data_set contains the calculated data (for now this is just the cmb velocity). The Data_Set class exists because the data in the Raw_Data_Set and Calc_Data_Set instances takes up a lot of memory, so the Data_Set class keeps them organized and saves/loads them to/from the hard drive to save RAM.

3.1 Data_Set Class

An instance of the Data_Set class can create a Raw_Data_Set, then save it to the hard drive and erase it from RAM to save space. It can also take the information in a Raw_Data_Set and use it to create a Calc_Data_Set instance, which it can then save to the hard drive and unload from RAM. Instances of the Calc_Data_Set can also load their corresponding Raw_Data_Set and Calc_Data_Set instances from the hard drive to access their data later.

Initializing a Data_Set takes a lot of arguments, so the simplest way to get your hands on one is to create an Analysis instance then use the Analysis.get_one_data_set() method, which takes no arguments and returns one Data_Set instance. Once you have a Data_Set, its Raw_Data_Set can be created by Data_Set.create_raw_data_set(), which takes one array and one constant as arguments. The array should have two columns: the first containing date times and the second containing z-positions. The constant specifies how many of the events are quip left (the first n_left rows of the array are assumed to be quip left). This can then be saved to the hard drive by calling Data_Set.save_raw_data_set(). Once the Raw_Data_Set is saved, it can be erased from memory by calling Data_Set.unload_raw_data_set() and then loaded again using Data_Set.load_raw_data_set(). Whenever the Raw_Data_Set is not loaded, Data_Set.raw_data_set is set to be an empty array, so you can check whether or not the Raw_Data_Set is loaded by calling isempty(Data_Set.raw_data_set).

The methods for the Calc_Data_Set are very similar, just change "raw" to "calc" in the method names. The one notable difference is that Data_Set.create_calc_data_set(), which is used to initialize the Calc_Data_Set, takes no arguments but can only be called after a Raw_Data_Set instance has been created.

3.2 Raw_Data_Set Class

This class stores the date times, charges, and z-positions for one data set. Each of these is separated into two arrays, one for quip left data and one for quip right data. A Raw_Data_Set instance can be accessed from a Data_Set instance as data_set.raw_data_set.

Iterating over quip left vs. quip right data can be annoying, so to make this simpler, the class includes two methods: get_data(direction_index) and get_date_times(direction_index). For each of these, a direction_index of 1 returns quip left data and direction_index 2 returns quip right data.

3.3 Calc_Data_Set Class

This class stores the data that can be calculated for each event from the data in the Raw_Data_Set instance. It was designed to store CMB velocities, moon positions, and sun positions, each in separate arrays for the quip left and quip right data. Currently only CMB velocities are stored. For easier access, this information can be retrieved from methods that take direction_index as an argument (again 1 for left or 2 for right). These methods are get_velocity(), get_moon_position(), and get_sun_position(), although only get_velocity() is implemented at the moment.

Chapter 4

Position_Generator Class

Need to write this.

Chapter 5

Random_Generator Class

Need to write this.

Chapter 6

Functions in Simulation-Data/Code/SignalFunctions

These functions are used to add signals to the raw data in data_set instances. Their names begin with "signal_" and they can be used to create Signal_Group instances in an Analysis instance. Each takes a data_set instance as an argument and does not return anything (the data_sets are modified in place). If you code your own signal function, I recommend basing it off of signal_sine, which is described below.

6.1 signal_null.m

This function does not touch the data in the data_set instance. It is by default the first signal_group in an Analysis instance's signal_group_list under the signal_name 'null'. A signal_group that uses this signal function can be added to an Analysis instance as demonstrated in the following example. Note that the signal_group will not be created if a signal_group with the given name or function handle is already in analysis.signal_group_list, and the 'null' signal_group is already included by default.

```
>> four_month.add_signal_group(@(data_set)signal_null(data_set),'null')
Warning: The given function name null is already in self.signal_group_list
> In Analysis>Analysis.add_signal_group at 202
```

6.2 signal_sine.m

This function takes four arguments. The first is the data_set, and the rest are the amplitude (in meters), period (in sidereal days), and phase (in radians). However, the signal function of a Signal_Group instance must take only one argument: the data_set. Therefore, to add a signal_group to an Analysis instance with this signal function, the given function handle must supply the rest of the arguments, as demonstrated in the example below.

```
>> four_month.add_signal_group(@(data_set) ...
signal_sine(data_set,0.05,'day',pi/8),'delayed sine');
```

Chapter 7

Functions in Simulation-Data/Code/OtherFunctions/

The SimulationData/ directory contains a mixture of data files separated into different folders. Because a lot of simulation data was generated[1], only small portions of each form of data are included in the Dropbox and Git repository. If you need larger portions of the data, contact Zak at uphgreat@gmail.com.

7.1 load/save_mat.m

These functions allow arbitrary Matlab objects to be saved and loaded. The function save_mat takes a file_name (which can include a path) and an object and saves the object with the given file_name. The load_mat function simply takes a file_name as an argument and returns the saved object.

```
>> some_array=rand(3)

some_array =

    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575

>> save_mat('some_file_name',some_array);
>> clear some_array
>> some_array=load_mat('some_file_name');
>> some_array

some_array =
```

[1] ∼400MB of time and position data was output by the simulations, which in turn is saved in many different ways, thereby increasing disk usage significantly.

```
    0.8147    0.9134    0.2785
    0.9058    0.6324    0.5469
    0.1270    0.0975    0.9575
```

7.2 filter_table.m

This function takes a table and unlimited filter pairs as arguments and returns a table in which some rows have been filtered out. The easiest way to understand it's behavior is through example. If you would like to only look at the rows in Charman_table where the quip is to the left and the data_set_index is 1, call the function as shown below. This function is used to take a large data table and pick out only the rows that you want.

```
>> Charman_table=four_month.signal_group_list{1}.Charman_table;
>> Charman_table(1:4,:)

ans =

    data_set_index    direction            A_1              abs_A_1
    --------------    ---------    -------------------    ---------

    1                 left              NaN+0i                 NaN
    1                 right        0.002504-0.0028301i    0.0037788
    1                 averaged     0.002504-0.0028301i    0.0037788
    2                 left              NaN+0i                 NaN

>> filtered_table=filter_table(Charman_table,'direction','right','data_set_index',5)

filtered_table =

    data_set_index    direction            A_1               abs_A_1
    --------------    ---------    --------------------    ---------

    5                 right        0.0010866+0.0018011i    0.0021035
```

Chapter 8

Functions in Simulation-Data/Code/ParameterFunctions

These functions take Data_Set instances and a direction as an argument and return an array giving the value of a parameter of interest at the event times of either the quip left or quip right data points. The argument direction is either 1 for left, or 2 for right; any other value causes an error. The names of each of these functions begins with "param_" to indicate their purpose. The first function of this type is param_speed() which returns the CMB speed of the Earth. Other similar function will be added in the future, such as the boost γ, angle between the trap and its CMB velocity, or parameters regarding the moon or sun position.

Chapter 9

Functions in SimulationData/Code/Mex/

This chapter covers the Mex functions in Simulation data that are used to interface with the Aephem library. It documents both how to compile the functions and how they are used by their corresponding Matlab wrappers.

9.1 Set-Up

After some code edits, the set up process has been simplified. If your machine is able to compile Mex files, all that needs to be done is to compile Aephem and put it in the correct directory. The code should be stored with the same directory structure as that of the Dropbox folder. After downloading the tarball for Aephem (I use version 2.0.0-Canopus, which is the latest version as of this writing), the tarball should be extracted into the aephem-2.0.0 directory. The directory name does not have any colons or spaces in it because this seems to cause trouble for the Mex compiler. Run 'make' to build the code. If you've already built the code, you can just move the directory containing the code to the appropriate place (in the same directory as SimulationData/) and rename it to "aephem-2.0.0". Or if you prefer not to have two copies of the software floating around, you might be able to get away with creating symbolic links.

If you have not used Mex on your machine before, it may need to be set up. I ran 'mex -setup' before I tried compiling anything, but this may not have been necessary. If you cannot build, refer to Matlab's documention for further instructions on getting Mex working.

9.2 moon/sun_position.c

These functions are intended to be compiled with Mex. After being successfully compiled, they take an array of Unix times and return three arrays giving the positions of the moon or sun: altitude angle (in degrees), azimuth angle (in degrees), and distance (in AU) in that order.

Because the input times are in Unix time, and because the libaephem library must be loaded before this function can run, wrapper functions datenum_to_moon_position() and

datenum_to_sun_position are included in the same directory. The wrapper functions load the libaephem library (if not already loaded), convert times from the datenum() format in Geneva time to Unix time, then call moon_position() or sun_position() and return the results. It is recommended that you use datenum_to_moon_position() and datenum_to_sun_position() instead of using moon_position() and sun_position directly.

Before moon/sun_position() or datenum_to_moon/sun_position() can be called, moon_position.c and sun_position.c must be compiled. The commands to compile them are included in the comments at the top of each function's source code and the commands are included below for reference. The commands can be run in Matlab and may also work in Bash. The scripts test_moon and test_sun will run these commands, so if you run those scripts you will not need run the Mex commands.

```
>> mex -O CFLAGS="\$CFLAGS -std=c99" ...
-I../../../aephem-2.0.0/src/ moon_position.c ...
../../../aephem-2.0.0/src/.libs/libaephem.so

>> mex -O CFLAGS="\$CFLAGS -std=c99" ...
-I../../../aephem-2.0.0/src/ sun_position.c ...
../../../aephem-2.0.0/src/.libs/libaephem.so
```

9.3 cmb_velocity.c

This function is intended to be compiled with Mex. After being successfully built, it takes an array of Unix times and returns three arrays, one for each velocity component. The velocities are given in m/s in J2000 cartesian coordinates.

Because the input times are in Unix time, and because the libaephem library must be loaded before this function can run, a wrapper function datenum_to_cmb_velocity() is included in the same directory. The wrapper function loads the libaephem library (if not already loaded), converts time from the datenum() format in Geneva time (it assumes Daylight Savings time effects have already been subtracted out) to Unix time, then calls cmb_velocity(), and finally return the results. It is recommended that you use datenum_to_cmb_velocity() instead of using cmb_velcoty() directly.

Before cmb_velocity() or datenum_to_cmb_velocity() can be called, cmb_velocity.c must be compiled. The command to compile it is included in the comment at the top of its source code and just below here for reference. The commands can be run in Matlab and may also work in Bash. The script test_cmb will run that commands, so if you run the test script you will not need run the Mex commands.

```
>> mex -O CFLAGS="\$CFLAGS -std=c99" ...
-I../../../aephem-2.0.0/src/ cmb_velocity.c ...
../../../aephem-2.0.0/src/.libs/libaephem.so
```

9.4 datenum_to_sun_position.m

This function takes times in datenum() format (given in Geneva time with Daylight Savings effects already removed) and returns the corresponding positions of the sun. It returns three arrays: altitude angle (in degrees), azimuth angle (in degrees), and distance (in AU) in that order. These values are calculated by the Mex function sun_position.c.

Because datenum_to_sun_position() uses a Mex Function, it requires some set up before it can be used, which is discussed in Chapter 9. Furthermore, because the Mex function uses Aephem, the library libaephem.so must be loaded. This generates several warnings, but these are inconsequential and so they are suppressed. If you would like to see these warnings, comment out the lines in the function that disable the warnings (these lines are easy to identify). Because the library is loaded only once per Matlab session, these warnings will only appear the first time you call a function that loads it in each Matlab session.

In order to facilitate testing of this function, a script with the name test_sun.m is included. This script will get four different times from generate_event_times(), compile the mex file from sun_position.c, then call datenum_to_sun_position() in order to test it.

```
>> test_sun %Input times are random, so your results will differ
Input times:
   1.0e+05 *

   7.348347028956812
   7.347951769621109
   7.348136403301930
   7.347713633204648

Altitude Angles (degrees)
 -33.523106831333557
 -48.426156349585526
 -44.265885795249218
 -44.483760867979612

Azimuthal Angles (degrees)
   1.0e+02 *

   2.737977248511227
   3.237035750192755
   2.974887156438333
   3.576023494954170

Distances (AU)
   0.986779021502397
   0.996202575443074
   0.991254353812390
   1.003038367934557
```

```
>> times=generate_event_times();
>> times=times(1:4); %shorten the data set for testing
>> disp(times); %datenum() format in Geneva time
   1.0e+05 *

   7.347261849032595
   7.347923670937909
   7.347966352360696
   7.348050297263487

>> [altitude_angles,azimuthal_angles,distances]=datenum_to_sun_position(times);
>> altitude_angles %Degrees

altitude_angles =

 -18.855431900039129
 -48.503379507476261
 -48.327077411463314
 -46.964272534957651

>> azimuthal_angles %Degrees

azimuthal_angles =

   1.0e+02 *

   0.403413145609679
   3.279057384436758
   3.215299601191740
   3.092656764031824

>> distances %Astronomical Units

distances =

   1.013595152713113
   0.996980530762410
   0.995803278933232
   0.993529922540726
```

9.5 datenum_to_moon_position.m

This function takes times in datenum() format (given in Geneva time with Daylight Savings effects already removed) and returns the corresponding positions of the moon. It returns three arrays: altitude angle (in degrees), azimuth angle (in degrees), and distance (in AU) in that order. These values are calculated by the Mex function moon_position.c.

Because datenum_to_moon_position() uses a Mex Function, it requires some set up before it can be used, which is discussed in Chapter 9. Furthermore, because the Mex function uses Aephem, the library libaephem.so must be loaded. This generates several warnings, but these are inconsequential and so they are suppressed. If you would like to see these warnings, comment out the lines in the function that disable the warnings (these lines are easy to identify). Because the library is loaded only once per Matlab session, these warnings will only appear the first time you call a function that loads it in each Matlab session.

In order to facilitate testing of this function, a script with the name test_moon.m is included. This script will get four different times from generate_event_times(), compile the mex file from moon_position.c, then call datenum_to_moon_position() in order to test it.

```
>> test_moon %Input times are random, so your results will differ
Input times:
   1.0e+05 *

   7.348301591664797
   7.347718621327302
   7.347453383644135
   7.348091955492030

Altitude Angles (degrees)
 -51.708581096340168
 -36.979561920095826
 -45.444516960693413
   9.860554431379413

Azimuthal Angles (degrees)
   1.0e+02 *

   3.135492160861178
   0.282011332586069
   0.119312144638621
   2.338023156631048

Distances (AU)
   0.002408605180408
   0.002422322724313
   0.002413445443471
   0.002588245543373
```

```
>> times=generate_event_times();
>> times=times(1:4); %shorten the data set for testing
>> disp(times); %datenum() format in Geneva time
   1.0e+05 *

   7.347261849032595
   7.347923670937909
   7.347966352360696
   7.348050297263487

>> [altitude_angles,azimuthal_angles,distances]=datenum_to_moon_position(times);
>> altitude_angles %Degrees

altitude_angles =

  -4.195638160911750
  32.117787548734356
  -9.564156988032691
 -34.946157346554727

>> azimuthal_angles %Degrees

azimuthal_angles =

   1.0e+02 *

   2.436268288734462
   0.912306838829998
   0.555622766047513
   2.732085666009608

>> distances %Astronomical Units

distances =

   0.002582072141966
   0.002682732729728
   0.002554265090355
   0.002424242457071
```

9.6 datenum_to_cmb_velocity.m

This functions takes times in datenum() format (given in Geneva time with Daylight Savings effects already removed) and returns the velocity of the Earth in J2000 cartesian coordinates. It returns three arrays, one for each component of the velocity, all of which are given in m/s. These values are calculated by the Mex function cmb_velocity.c.

Because datenum_to_cmb_velocity() uses a Mex Function, it requires some set up before it can be used, which is discussed in Chapter 9. Furthermore, because the Mex function uses Aephem, the library libaephem.so must be loaded. This generates several warnings, but these are inconsequential and so they are suppressed. If you would like to see these warnings, comment out the lines in the function that disable the warnings (these lines are easy to identify). Because the library is loaded only once per Matlab session, these warnings will only appear the first time you call a function that loads it in each Matlab session.

In order to facilitate testing of this function, a script with the name test_cmb.m is included. This script will get four different times from generate_event_times(), compile the mex file from cmb_velocity.c, then call datenum_to_cmb_velocity() in order to test it. It also calculates the magnitudes of the resulting velocities.

```
>> test_cmb %Input times are random, so your results will differ
converting times
Elapsed time is 0.000292 seconds.
velocity calculation
Elapsed time is 0.001435 seconds.
Input times:
   1.0e+05 *

   7.348347028956812
   7.347951769621109
   7.348136403301930
   7.347713633204648

Velocity x-components (m/s)
   1.0e+05 *

  -3.864509329877444
  -3.721512930742001
  -3.800308685434046
  -3.603658673235534

Velocity y-components (m/s)
   1.0e+05 *

   0.887812105085747
   1.019058074180573
   0.969425092132093
```

```
   1.045073555101387

Velocity z-components (m/s)
   1.0e+04 *

  -3.944129486583106
  -3.374932025980408
  -3.590177311038611
  -3.262225784947812

Speeds (m/s)
   1.0e+05 *

   3.984746449740051
   3.873246637387350
   3.938403844844726
   3.766292041182909

>> times=generate_event_times();
>> times=times(1:4); %shorten the data set for testing
>> disp(times); %datenum() format in Geneva time
   1.0e+05 *

   7.347261849032595
   7.347923670937909
   7.347966352360696
   7.348050297263487

>> [v_x,v_y,v_z]=datenum_to_cmb_velocity(times);
converting times
Elapsed time is 0.000996 seconds.
velocity calculation
Elapsed time is 0.000386 seconds.
>> disp(v_x)
   1.0e+05 *

  -3.395412720635101
  -3.708260010759425
  -3.728301051852960
  -3.765806260905132

>> disp(v_y)
   1.0e+05 *

   0.974816115275701
```

```
    1.024361870822486
    1.016070340793857
    0.995627080751760

>> disp(v_z)
    1.0e+04 *

   -3.566948235152507
   -3.351925667711104
   -3.387909007399682
   -3.476737793988580
```

Chapter 10

Data Directories in SimulationData/DataSets/

As of the moment that this is being written, we have not agreed upon a way to choose the times of the events for the simulated data sets. Therefore I plan on creating multiple groups of data sets, each with a different version of generate_event_times(). This chapter includes descriptions of these different groups of data sets.

10.1 FourMonth

This data was generated with random times. The distribution function was chosen to be uniform over the period from 8/1/2011 to 12/1/11. This was chosen to be very roughly the time of the year that data was collected in 2010 and 2011.

Chapter 11

Functions in Simulation-Data/CorrelationFunctions/

This folder contains the functions used to implement Andy Charman's algorithms. His descriptions/derivations of these algorithms are described in Charman_sinusoid_estimator.pdf, which is included in the folder with this document.

There is an important distinction between a Matlab day and a sidereal day. A Matlab day is defined as $60 * 60 * 24 = 86,400$ seconds while a sidereal day is how long the Earth takes to revolve once with respect to the distant stars, which is 86,164.09 seconds. To make the code more straight forward, all the data is stored in terms of Matlab days (because this is what datenum() uses). However, we are more interested in sidereal days when it comes to periodic signals, so the period arguments are given in sidereal days for convenience and then they are converted inside the function to Matlab days for calculations.

11.1 CharmanII.m

This function returns the complex Fourier Coefficient A_1, given three arguments: date_times, data, and period. The date_times argument should be an array giving the date times of the events. The data argument should be an array of the same dimensions as date_times giving either the z-positions or wait times of the annihilations. The period should be 'day', 'month', 'year', or a time in units of sidereal days.

This function uses the method described in Section II of Charman_sinusoid_estimator.pdf. It is best suited for period='year' because it is not expected to be as powerful as CharmanIV.m for daily or monthly signals, but does not need the data to be well-distributed across the time period. If the input data array is empty, this function returns NaN.

11.2 CharmanIV.m

This function returns the complex Fourier Coefficient A_1, given three arguments: date_times, data, and period. The date_times argument should be an array giving the date times of the events. The data argument should be an array of the same dimensions as date_times giving

either the z-positions or wait times of the annihilations. The period should be 'day', 'month', 'year', or a time in units of sidereal days.

This function uses the method described in Section IV of Charman_sinusoid_estimator.pdf. It is best suited for period='day' because it requires the data to be well spread throughout the period and assumes a fixed frequency. If the input data array is empty, this function returns NaN.

Chapter 12

Miscellaneous

This chapter is intended to be a place for information that doesn't quite fit anywhere else.

12.1 SimulationData/TracerOutput/

This folder contains the end results of the $\bar{H}$ tracer simulations and this data is used to generate all the simulated data sets. The data is stored in two different ways: .mat and .ellip files. The .ellip files are text files and the .mat files are Matlab binary files that are smaller and can be read/written faster than the .ellip files. Therefore, only use the .ellip files for visual inspection with a text editor.

Each .ellip file in this folder has two columns. The first column is the wait time it took (in seconds) for the $\bar{H}$ to hit the trap walls after the quench. Note that this is *not* the wall time of the event. The second column is the z-position (in meters) of the annihilation with respect to the trap center. This set of data includes effects from detector smearing and cuts[1]. This data is read in automatically by instances of the Position_Generator class when generating Raw_Data_Set instances.

Currently three tracer data sets are included in this directory: MediumSimData, LargeSimData, and AllSimData. MediumSimData and LargeSimData contain a portion of the full set of simulation output AllSimData, which is ∼400MB so it is not included in the Dropbox folder. Each of these has two versions: one in which the rows are randomly oriented (whatever order that they came out of the tracer simulations) and one in which they are sorted by z-position. The sorted versions are used in case a future version of the Position_Generator class does some interpolation between these event positions.

[1] Cuts are criteria used to distinguish $\bar{H}$ annihilations from other events, such as those due to cosmic rays.