Code Documentation

Fajans Lab and & Friends

January 13, 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. When working on it, edit a local copy on your machine (not saved in your Dropbox folder) so that if anyone's work gets overwritten, they still have a copy of it. When you add a section, make sure you add it to the most recent version so it includes all the latest additions.

I'm not sure what the best way to organize this document is, but for now I think it will suffice to make a new chapter for each category of functions (parsing elogs, calculating positions of heavenly bodies, etc.), each of which should have its own folder in the Dropbox. Within each chapter, create a section for each function that describes its inputs, outputs, algorithm, example usage, and compilation instructions if necessary. We should also develop some kind of naming scheme (for example camelCase or names_with_underscores) at least for function names, and ideally for code within functions. This should also be recorded here.

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. 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. To make a deep copy of an instance, use its deep_copy() method[1]. As of this writing, the code is still in development, so in the interest of time, it does not run on the full tracer data set. Instead, it only uses the data in LargeSimData.met by default. To switch use all of the tracer data, run analysis.set_tracer_file('all') after initializing your Analysis instance and before generating the raw_data_sets.

The examples in this section were run on the data from LargeSimData.mat, so don't be surprised if you are working with AllSimData.mat and the function take much longer to run.

1.1 Set-up/Initialization

The first step to create an Analysis instance is to create a directory in SimulationData/DataSets/. The name of this directory will be used to identify the generate_event_times() function, so give it a name that describes how that function picks times. For example, the first one I made randomly picks times over a four month period, so I created a directory and called it FourMonth. Place the associated version of generate_event_times.m in this directory and make sure no other versions of that function appear in your path.

Now you are ready to initialize an instance of this class. To do this, change to the directory[2] SimulationData/code/. 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

[1]Note that this function no longer works. It stopped working around the time I upgraded to Matlab 2013b from 2013a, so it may have to do with that. It doesn't work because it tries to assign to the read-only properties of the Analysis class.

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

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 often automatically called internally). This is demonstrated in the example below; notice how four_month.Charman_table is already full of data as soon as four_month is initialized the second time.

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

four_month =

  Analysis with properties:

    TRACER_FILE_NAME: '../TracerOutput/LargeSimData.mat'
      GENERATOR_NAME: 'FourMonth'
       data_set_list: {}
       Charman_table: []
       data_set_root: '../DataSets/FourMonth'

>> four_month.run()

----blah, blah, blah.  Lots of output----

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

four_month =

  Analysis with properties:

    TRACER_FILE_NAME: '../TracerOutput/LargeSimData.mat'
      GENERATOR_NAME: 'FourMonth'
       data_set_list: {}
       Charman_table: [24000x6 table]
       data_set_root: '../DataSets/FourMonth'
```

1.2 Analysis.run()

This is the most high-level function in the class. It automatically runs all of the tasks to perform the analysis and displays information showing the progress of the calculations. Simply create an instance of the analysis class and call analysis.run() to generate all the data. As more methods are written for the class, more steps will be added to this method. Right now it runs Analysis.generate_raw_data_sets(), Analysis.generate_calc_data_sets(), Analysis.generate_Charman_tab and Analysis.generate_Charman_histograms().

1.3 Analysis.generate_raw_data_sets()

This method takes no arguments and creates pseudo data sets by pairing wait times and z-positions from the tracer output with date times from the generate_event_times() function in the analysis folder (Analysis.data_set_root). Each data set has 386 events, of which the first 145 are assumed to be quip left data. Raw_Data_Sets are produced continuously until all the points in the tracer data are used up. Each Raw_Data_Set instance is attached to a Data_Set instance[3], which handles loading and unloading the Raw_Data_Set instances to/from the hard drive in order to minimize RAM usage.

Because generating all the Raw_Data_Sets can take a long time, the code is written to run in parallel. However, to do this, the parallel processing toolbox must be installed. If you do not have the parallel processing toolbox, the code may not run. If this is the case, you may need to remove the lines that call the matlabpool function and change all the "parfor" loops to "for" loops. After that, the code should run just fine.

If a pool of workers is not running before this function is called, it will automatically open a pool, which will take an additional ∼10 seconds.

```
>> matlabpool('open')
Starting matlabpool using the 'SevenLocalWorkers' profile ... connected to 7 workers.
>> four_month.generate_raw_data_sets()
Generating raw data sets
Number of events from Tracer data: 386000
Number of events per data set: 386
Finished generating raw data sets
Generated 1000 raw data sets
Generating raw data sets took 5.09 seconds
```

1.4 Analysis.generate_calc_data_sets()

This method takes no arguments and generates Calc_Data_Sets for each Data_Set. These Calc_Data_Sets are designed to hold all the values returned by datenum_to_cmb_velocity(), datenum_to_moon_position(), and datenum_to_sun_position() for the date times in each Data_Set. Currently, the instances only hold the results from datenum_to_cmb_velocity(), but this should change in the future.

This process can be very time-consuming, so it is designed to run in parallel. It's behavior with respect to parallelization is the same as Analysis.generate_raw_data_sets(), so refer to that method's documentation for further details.

This function uses the data stored in the Raw_Data_Sets, so make sure to run Analysis.generate_raw_data_sets() before calling this method.

```
>> four_month.generate_calc_data_sets()
Generating calculated data sets
Finished generating calculated data sets
Generating calculated data sets took 11.92 seconds
```

[3]This is a good time to take a quick look at Chapter 2 to see how this data is stored

1.5 Analysis.generate_Charman_table()

This method takes no arguments and generates a table that contains all the results of applying CharmanII() and CharmanIV() to the generated Raw_Data_Sets. This table is stored as analysis_instance.Charman_table. Because it uses the Raw_Data_Sets, make sure to call Analysis.generate_raw_data_sets() before using this method. This method also has the same parallelization behavior as Analysis.generate_raw_data_sets(), so refer to that section for more information.

```
>> four_month.generate_Charman_table()
Generating Charman table
Finished generating Charman table
Generating Charman table took 5.87 seconds
>> disp(four_month.Charman_table(1:4,:))
    data_index    data_type     algorithm     period    direction             A_1
    ----------    ----------    ----------    ------    ---------    -------------------

    1             z-position    Charman II    day       left          0.0017035+0.004843
    1             z-position    Charman II    day       right         -0.004411+0.004282
    1             z-position    Charman II    day       averaged     -0.0021141+0.004493
    1             z-position    Charman II    year      left           0.045676+0.029494
```

1.6 Analysis.generate_Charman_histograms()

This method produces a few histograms that appear as docked figures. It can be called with one argument, which will set the number of bins in the histogram, or it will default to 25 bins if no argument is provided. The most you can reasonably have with the data from LargeSimData is ∼25, 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()
Generating Charman histograms
Finished generating Charman histograms
Generating Charman histograms took 0.47 seconds
```

Chapter 2

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. This is done because the data in the Raw_Data_Set and Calc_Data_Set instances takes up a lot of RAM, so the Data_Set class keeps them organized and manages them.

2.1 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, it's Raw_Data_Set can be created by Data_Set.create_raw_data_set(), which takes one array as an argument. The array should have three columns: the first containing date times, the second containing wait times, and the third containing z-positions. 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.

2.2 Raw_Data_Set Class

This class stores the date times, wait times, 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 and over wait times vs. z-positions can be annoying, so to make this simpler, the class includes two methods: get_data(data_index, direction_index) and get_date_times(direction_index). The get_data method takes two indices as arguments. If the data_index is 1, it returns z-positions, if the data_index is 2, it returns wait times, and it errors on any other value. If the direction_index is 1 it returns quip left data, if it is 2 it returns quip right data, and it errors on any other value. The get_date_times() method only takes the direction_index since it doesn't have to choose between z-positions or wait times.

2.3 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. This means it stores CMB velocities, moon positions, and sun positions, each stored in separate arrays for the quip left and quip right data. 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()[1].

[1] At the moment only get_velocity() is implemented.

Chapter 3

Matlab Functions in SimulationData/Code/

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. If you need larger portions of the data, contact Zak at uphgreat@gmail.com.

3.1 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
```

3.2 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 algorithm is Charman II, the period is one day, and the quip is to the left, 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.Charman_table;
>> Charman_table(1:4,:)

ans =

    data_index    data_type     algorithm     period    direction             A_1
    ----------    ----------    ----------    ------    ---------    -------------------

    1             z-position    Charman II    day       left         -0.0015231-0.004259
    1             z-position    Charman II    day       right         0.0022921+0.000293
    1             z-position    Charman II    day       averaged     0.00085891-0.001416
    1             z-position    Charman II    year      left           0.034328+0.010111

>> filtered_table=filter_table(Charman_table,'algorithm','Charman II', ...
    'period','day','direction','left');
>> filtered_table(1:4,:)

ans =

    data_index    data_type     algorithm     period    direction             A_1
    ----------    ----------    ----------    ------    ---------    -------------------

    1             z-position    Charman II    day       left          -0.0015231-0.00425
    1             wait_time     Charman II    day       left         -3.1669e-05-0.00038
    2             z-position    Charman II    day       left          -0.0039187+0.00406
    2             wait_time     Charman II    day       left         -0.00032895+9.1809e
```

3.3 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 4. 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
```

3.4 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 4. 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
```

3.5 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 4. 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
```

3.6 Parameter Functions

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, or parameters regarding the moon or sun position.

Chapter 4

Mex Functions in SimulationData/Code/

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.

4.1 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.

4.2 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
```

4.3 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
```

Chapter 5

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.

5.1 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 6

CharmanUltra/

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 some arguments are given in sidereal days and converted inside the function to Matlab days for calculations.

6.1 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.

6.2 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.

Chapter 7

Miscellaneous

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

7.1 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 Analysis 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.

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