DESCRIPTION OF THE FRACTIONAL CHARGE SIMULATION CODE

MARCELO BAQUERO-RUIZ

1. Overview

The fractional charge simulation code is a program in C++ that integrates the full 3D trajectories of charged particles in the ALPHA1 trap starting from random initial conditions generated using an experimentally motivated distribution. It is based on a code developed originally by Prof. Francis Robicheaux; that means that there are many things (including the integration scheme) which were inherited from his original version.

The code includes interactions with the full magnetic field produced with the mirror coils, octupole, and background 1T solenoid, and with the electric fields produced with biases at the electrodes. Furthermore, it provides a way to simulate manipulations of the electric field, such as the ones performed during usual trapping attempts in ALPHA.

The fields produced by the mirrors and octupole are implemented following the models presented in [1]. The background field is constant in value and hardcoded in the program. The time evolution is simple, and changes only happen after the trap shutdown trigger; the magnetic field then follows an approximately exponential decay, whose precise form is implemented using an interpolation on a set of points calculated by Joel for the mirrors and octupole (which includes the effects of eddy currents). This set of points is read from a text file that must be accessible by the main program (see sec. 4). All other functions related to the evaluation of the magnetic fields are included in the code, and any modifications can be made by changing parameters therein.

A full, time-dependent, 3 D electric field is implemented that allows to calculate the field anywhere in the trapping region at any moment of the sequences. This is accomplished by, first, creating a dense grid of points that covers the whole volume and evaluating the value of the electrostatic potential in each point of the grid for each different set of applied voltages at the electrodes. Sets of splines are then created to interpolate the potentials on the grid for each configuration of electrode biases. The time dependency is finally added with functions that make transitions between the different sets of spline coefficients.

Even though most of the electric field steps are implemented in the main C++ code, the files containing the sets of spline coefficients must be generated with a different (separate) program. This will be explained in more detail in sec. 3.

The initial conditions of the particles are generated randomly using models for the initial volume and initial energy distributions, which can be altered in the code by using different parameters or changing the distributions themselves. However, it is important to note that an external seed must be provided for the random number generator (this is done to be able to run many parallel instances of the program without getting duplicated results). Usually, this is done by passing the value of the seed as an argument to the executable (see sec. 4).

2. The Code - parameter setup

The main code, *frac_charge.cpp*, is very self-contained. No libraries (other than standard ones) or different pieces of code need to be linked in order to compile it and run it.

Date: September 27, 2013.

Thanks to Zak Vendeiro for helpful comments.

There are, however, some files that need to be accessed by the executable, and stored *in the same directory* where it runs:

- The file *QuenchData.dat*, which holds the information for the magnetic field decay after shutting down the magnets. This file should in principle never be changed or updated unless trap decay profiles different from the "true" ALPHA1 ones need to be modeled.
- The files *SpCoefChuk_z2_r128_*.bin* that contain the interpolation coefficients for the different electrode biases and that need to be generated with a separate program (see sec. 3)

For reference, all these files have been copied to the following directory in the Dell lab server (using the *fajanslab* account):

`/home/fajanslab/marcelo/Frac_Charge/bkup/code/`

With the exception of the seed of the random number generator, all the other parameters used in *frac_charge.cpp* are hardcoded and need to be changed directly in the code whenever simulations with different values want to be performed. This obviously implies that a new compilation is required after each modification (see sec. 4). The number of relevant parameters, however, is small. They appear in few parts, as described in the algorithm of the program:

(1) Define fundamental constants and global variables.
(2) Once in the main, start by defining the very important parameters:
    - *tfin*: Maximum simulation time for each particle trajectory (default 190 ms)
    - *tqu*: The time of the magnet shutdown trigger (default 147.3 ms).
    - *dtim*: The time step (default 3.5 μs).

    These numbers should not be changed unless manipulations different than the ones in the nominal trapping sequences need to be simulated.
(3) Read the *seed* for the random number generator from the list of arguments passed to the program, and initialize the generator.
(4) Load the spline coefficients for the electrostatic potentials (mentioned above) using the function *LoadCoeficients()*. The names of the binary files that hold the interpolation coefficients must be correctly written in that function.
(5) Initialize values of other parameters used in the program. The most important ones (the others will not need to be changed unless a substantial change in the model occurs) can be found inside the function *InitialSetup()*:
    - *fracq*: The value of fractional charge Q considered.
    - *numtraj*: Number of different particle trajectories to simulate; these are done sequentially: After reaching the end of a simulation, a new one starts from the beginning by generating new random initial conditions.
    - *ioct*, *bsol1*, *bsol2*: Currents (in A) through the octupole (default 885.7 A) and mirrors (default 626.5 A), respectively. *bunif*: background magnetic field (in T; default 1 T).
(6) Load data for the magnet decay curves with *LoadQuenchData()*.
(7) Generate an initial simulation time of ≈ -1 s, and reset potentials to their initial value. The extra ≈ 1 s gives time for the particle trajectory to be properly randomized before the start of the potential manipulations (which occurs at $t = 0$).
(8) Generate initial conditions for the new trajectory to be simulated. The initial position and velocity are randomly generated using the function *GenInitCond()* which, in the case of a Maxwellian distribution, uses the following important parameters:
    - *Tdist*: Temperature of the distribution of initial particle energies (in K; default is 50 K).
    - *Tcut*: Minimum "cutoff" energy at which antihydrogen can no longer be confined in the trap (in K; default is 0.75 K). One would in general want to set this value to be

larger than the trap depth, which will vary depending on the values chosen for the mirrors and octupole fields.

(9) Step forward in time. The integration scheme uses a symplectic stepper implemented with the function *Symplec_mod()*.
The time dependence of the magnetic and electric fields is included in *Symplec_mod()* through the functions *UpdateCurrents_mod()* and *PotentialTransitions()* respectively. The forces are calculated with the use of *ForceB()* and *ForceE()* plus the calculation of the variable array *vec[]*.

(10) Check whether the particle is still in the trapping region. If yes, go back to step (9). Otherwise, the simulation of this particular trajectory is over. Print t, the simulation time at which the particle "escaped"; and the axial coordinate z (the annihilation location). Continue to the next step.

(11) Check whether the total number of trajectories desired is complete. If not, go to step (7); otherwise, finish.

In the end, a list of *numtraj* pairs (t, z) is printed on the screen that shows the times and axial locations of particles that come into contact with the electrodes. In the case of antihydrogen atoms, these are simulated annihilations.
Two important comments are in order:

- The times t in the list are the annihilation times with respect to the start of the manipulations. In order to have these times be referenced with respect to the trigger of the magnetic trap shutdown (which is usually what is important for the different analyses), the quantity *tqu* must be subtracted.
- The locations z are referenced with respect to E10 (electrode 10 in the stack). In order to have z be referenced with respect to the axial center of E18, which is the convention followed in the fractional charge paper and in [2], 183.385 mm must be subtracted from the values of z in the list.

3. Generation of electric potential maps

As mentioned before, the electric field calculations in *frac_charge.cpp* employ separate sets of interpolation coefficients for the potentials that need to be generated in advance by a different program. Those coefficients are used to interpolate the potential in a grid of the following characteristics:

- 2001 points in z (axially) in the range $-183.385\,\text{mm} \leq z \leq 195.10\,\text{mm}$. Here z is given assuming that $z = 0$ happens exactly at the axial center of E18 (electrode number 18 in the ALPHA1 stack).
- 129 points in r (radial distance from the axis of the trap) where $0 \leq z \leq 22.275\,\text{mm}$.

Note that, since some electrodes far from $z = 0$ have a smaller radius, some of the points covered by this grid are outside the region of interest. This is of no consequence, since the simulations stop once anti-atoms leave the trapping region.

The way this was done for the standard simulations used *make_splines_chuk.cpp* to read a set of files generated by Chukman So, which contained the potential evaluated on the aforementioned grid using his ALPHA1 trap model in COMSOL (for each electrode bias setup), calculate the corresponding spline coefficients, and store them in binary files that have the appropriate format that can be understood by *frac_charge.cpp*. Therefore, this approach required sending an email to Chukman, stating the required biases on the electrodes for the different configurations of interest, and kindly asking him to obtain the lists of potentials.

One important comment, though. Some of the electrodes in ALPHA1 were "slow", meaning that they were more heavily filtered to reduce electrical noise in sensitive areas of the experiment, thus having slower response times to changes in the biases. This means that, in order to correctly model

transitions, different sets of interpolation coefficients had to be created for the "slow" electrodes (keeping all others at 0 V) and for the other "fast" electrodes (keeping the "slow" electrodes at 0 V); the total potential and/or electric field could be found in the main program by superposition. This is important because both files need to be created and opened in *make_splines_chuk.cpp*, but only one binary file (that contains information for both "fast" and "slow" electrodes) per configuration is created that is later used by *frac_charge.cpp*.

There is another approach that does not require bugging Chukman. The program *make_splines.cpp* calculates the potentials in the grid from values of the electrode potentials hardcoded in the function *InitialSetup()*, for both "slow" and "fast" electrodes, creates the spline coefficients, and automatically creates the binary files in the correct formats. Why not use this procedure always instead? Because the potential solver is much cruder. It does not take into account the change in electrode radius, and does not model the gaps between electrodes correctly. However, it can be used whenever a quick raw calculation is needed.

4. Compilation and execution

As mentioned above, the code is very well contained. It does not need anything fancy or extra complications for its compilation. It does, however, require a C++ compiler (only C will not work). From a standard BASH shell (found in any Linux machine) simply issue:

```
g++ frac_charge.cpp -o frac_charge.exe -Wall
```

To run the executable, remember that *QuenchData.dat* and all *SpCoefChuk_z2_r128_*.bin* files must be in the same directory as the ".exe". Also, remember that one argument needs to be passed to set the seed of the random number generator. Simply issue

```
./frac_charge.exe 1
```

or replace "1" with any positive integer to change the seed.

In that form, the output will be shown on the screen. If the numbers want to be stored in a file, they can be easily redirected by doing instead

```
./frac_charge.exe 1 > file_name.dat &
```

where "file_name.dat" can be any name of your choosing. Please remember that redirecting the output in this way will overwrite any existing data in "file_name.dat". The & at the end can be removed if the process does not want to be sent to the background.

A similar procedure can be done whenever a system with installed parallel MPI capabilities is used, such as the Dell server in the lab running under Linux. In that case, the main program *frac_charge.cpp* has to be slightly modified to include the appropriate parallelization instructions. This is exactly what was done to create *frac_charge_mpi.cpp*. The compilation then proceeds as

```
mpic++ frac_charge_mpi.cpp -o frac_charge_mpi.exe -Wall
```

Execution also requires having in the same directory the binary files with the potential spline coefficients, and the magnet shutdown data. A seed must also be provided as an argument, and additionally, one needs to specify the number of parallel processes to use (using the option "-np"). For example, if one wants to run a code that uses the seed "5" and runs 62 different parallel instances (something that can be done in the Dell lab server with 64 cores), one only needs to issue the command:

```
mpirun -np 62 frac_charge_mpi.exe 5 &> file_name.dat &
```

It is important to mention that the parallel instances use different seeds generated from the base one passed as an argument. Thus, if additional simulations need to be run, the new seed passed as an argument must have a value of at least the seed used in the previous run *plus* the number of parallel instances selected in order for the results not to clash with the previous ones.

5. Adding detector effects and cuts

Detector effects, as described in more detail in [2] comprise the addition of smearing and detector efficiency in the axial direction z. The "cuts" are just a selection of data following a pre-established criterion.

In the case of the simulation results, detector effects and cuts are added with the program *apply_ZeffANDsmear.cpp*, which takes as an input the text file with annihilation times and locations obtained with *frac_charge.cpp*, and produces a text file with a new list of annihilation times and locations that includes the changes.

apply_ZeffANDsmear.cpp offers the possibility to choose different detector efficiency models by changing the argument passed to the function *CreateSplinesEff()*; the list of possible models is clearly indicated inside that function.

Once the different parameters are set (hardcoded), the C++ code needs to be compiled in a way similar to what was described above (no extra libraries or files need to be linked):

```
g++ apply_ZeffANDsmear.cpp -o apply_ZeffANDsmear.exe -Wall
```

The executable needs the name of the file with the raw list of simulated annihilations to be passed as an argument. It then creates a file with a similar name but a modified extension where the output data is stored:

```
./apply_ZeffANDsmear.exe raw_data.dat
```

In this case, the file "raw_data.test" would be created.

If only the mean of the distribution and its error are needed, another program can be used that does not create any additional files, *mean_ZeffANDsmear.cpp*. The numbers calculated include cuts and detector effects, so it provides a better way to extract the information that one is usually interested in the most for the analyses of fractional charge. It is written pretty much in the same way as *apply_ZeffANDsmear.cpp*, uses the same parameters, and allows the use of different efficiency curves as well. Once compiled, it can be run by doing:

```
./mean_ZeffANDsmear.exe raw_data.dat
```

The results will be shown on the screen.

6. Stochastic heating code

The simulations of stochastic heating use a code, *frac_stoc_heat.cpp*, very similar to the one described in sec. 2. However, some of the complexity required for the fractional charge simulation is no longer needed and many of the functionalities in *frac_charge.cpp* are not used. For example, simulations do not need to model the magnetic trap shutdown, since only the number of surviving particles, and not their precise location, is of interest. Consequently, the program does not need to load the data in the file *QuenchData.dat*.

Also, the electrode biases are simpler, and no ramps between different configurations are needed; thus, no transitions to different sets of potential interpolation coefficients need to be implemented.

Nevertheless, the program requires the use of one file of spline coefficients that needs to be generated in advance following the same procedure as described in sec. 2. That file, named *SpCoefChuk_z2_r128_StocHeat.bin*, contains the splines for the electrostatic potential generated by electrode biases of 1 V, −1 V, 1 V, etc (alternating between consecutive electrodes).

The parameters used to set the total simulation time, fractional charge, number of trajectories to simulate, magnetic field strengths and initial conditions are the same as the ones described in sec. 2. Here is a summary of the most important parameters that need to be set directly in the code (hardcoded) for the simulations:

- *tfin*: Maximum simulation time for each particle trajectory (can be as high as 10 000 s or more).
- *dtim*: The time step. The is default 35 μs, which is larger than before in order to simulate the extremely long heating times of interest.
- *fracq*: The value of fractional charge Q considered.
- *numtraj*: Number of different particle trajectories to simulate.
- *ioct*, *bsol1*, *bsol2*: Currents (in A) through the octupole (default 885.7 A) and mirrors (default 626.5 A), respectively; these values can be changed to produce shallower magnetic traps when needed. *bunif*: background magnetic field (1 T).
- *Tdist*: Temperature of the distribution of initial particle energies (in K).
- *Tcut*: Minimum "cutoff" energy at which antihydrogen can no longer be confined in the trap (in K). As before, one would in general want to set this value to be larger than the trap depth, which will vary depending on the values chosen for the mirrors and octupole.

Additionally, there are in these case other parameters needed for configuring the stochastic heating drive. As mentioned in [2], the biases at the electrodes switching between "+" and "-" randomly following a uniform distribution.

- *sh_ampl*: Amplitude of the stochastic drive (V).
- *t_mean*: Mean time length between stochastic heating transitions (s).
- *t_sdev*: Time corresponding to 1 std.dev. of the stochastic transition timings.

As with the other programs mentioned earlier, the compilation is straightforward and does not require linking any additional pieces of code. Once all the parameters have been hardcoded and saved, just do

```
g++ frac_stoc_heat.cpp -o frac_stoc_heat.exe -Wall
```

To run the executable, remember that the file *SpCoefChuk_z2_r128_StocHeat.bin* must be present in the same directory. Also, remember that a seed must be passed as an argument to initialize the random number generator. Just do:

```
./frac_stoc_heat.exe 1
```

where a seed of value "1" was chosen. A list with information about the particle that left the trap before the end of the simulation, or that survived all the way until the end, is printed on the screen for each trajectory.

As before, the value of the seed can be changed to any positive integer, a feature that is useful when running many instances of the program in parallel.

References

[1] C. Amole et al. (ALPHA Collaboration), *Discriminating between antihydrogen and mirror-trapped antiprotons in a minimum B-trap*, New J. Phys 14, 015010 (2012).

[2] M. Baquero-Ruiz, *Studies on the neutrality of antihydrogen*, Ph.D. Thesis, University of California at Berkeley (2013).