2+1-dimensional Fixed Boundaries CDT: Users Guide

Jonah Miller
jonah.maxwell.miller@gmail.com

August 26, 2012

1 Introduction

This is the fixed boundaries branch of Rajesh Kommu's CDT code. The algorithm was originally worked on by David Kemansky, and later improved upon and updated by Jonah Miller.

The code implements CDT using an updated action that includes the boundary terms.

The goal of this file is to explain to the educated user how to run a simulation. I will focus only on the basics of the user interface for the LISP code. I have added a large number of python and shell scripts as well, which I will not discuss in detail here. Each script should be well commented, and there should be a brief description of use in the file called "program_and_module_list.txt". If you have any questions, please feel free to email me.

2 Running the Code

The code is typically run by creating a Common Lisp script and running it using the command

```
nohup nice sbcl --dynamic-space-size 2000 --script <your-script> >> logname.log &
```

In order, each command does the following:

- nohup—stands for "no hangup." Prevents the program from stopping if you log off of your computer.
- nice—Tells the computer to prioritize this program less highly than other programs you start. What this means is that programs without the "nice" flag will take more cputime than programs that have the "nice" flag. This prevents the computer from freezing up if all the cores are running simulations.
- sbcl—Steele Bank Common Lisp. The Common Lisp interpeter we use.
- dynamic-space-size 2000— This tells the lisp interpreter to reserve 2 GB of RAM for use with the garbage collector. If the we don't reserve enough memory, the program crashes.
- script [your script name]—This is hopefully self-explanatory.
- >> logname.log—This sends the output of the program from the console to the file logname.log.
- &—This "daemonizes" the program, so you can still use your console.

It is also possible to run lisp in a live interpreted environment using common lisp's REPL (read-eval-print-loop), which compiles individual functions as they're defined and runs them in an interpreted environment. This is very nice for debugging and prototyping. To do this, just type

```
sbcl
```

into the command line.

Alternatively, you can use SLIME, an extension for emacs, as a more powerful interpreter and indeed IDE. Slime is a very powerful tool and I suggest you check it out:

`http://common-lisp.net/project/slime/`

3 Dependencies

One of the first lines of your script should be

```
(load ``cdt2p1.lisp'')
```

This is a top level wrapper which includes all the most important modules. It is equivalent to:

```
(load "../utilities.lisp")
(load "globals.lisp")
(load "tracking_vectors.lisp")
(load "action.lisp")
(load "reset_spacetime.lisp")
(load "generalized-hash-table-counting-functions.lisp")
(load "simplex.lisp")
(load "topological_checks.lisp")
(load "moves.lisp")
(load "ascii_plotting_tools.lisp")
(load "initialization.lisp")
(load "montecarlo.lisp")
(load "output.lisp")
```

4 Getting a spacetime

When you run a simulation, you're going to want to do one of two tings: create a new spacetime to thermalize, or load an existing spacetime from a file.

4.1 Creating a new spacetime

You can initialize your spacetime with the "set-t-slcies-with-v-volume" command. For example:

```
(set-t-slices-with-v-volume :num-time-slices 64
                            :target-volume 30850
                            :spatial-topology "S2"
                            :boundary-conditions "OPEN"
                :initial-spatial-geometry "boundary_files/tetrahedron.boundary"
                :final-spatial-geometry "boundary_files/tetrahedron.boundary")
```

Some options require explanation.

- The input number of time slices *must* be even, due to the way the simulation initializes the spacetime. In the periodic boundary conditions case, the actual number of time slices the computer will generate is indeed the number you put here. However, in the fixed boundary conditions case, one additional time slice will be added.
- The options for spatial topology are "S2" and "T2," for the sphere and the torus respectively. The sphere topology works for everything. However, the torus only works for periodic boundary conditions.

- Boundary condition types are "OPEN" and "PERIODIC".
- The "initial-spatial-geoemtry" and "final-spatial-geometry" fields are entirely optional. If you don't want them, don't even type them. Like so:

```
(set-t-slices-with-v-volume :num-time-slices 64
                            :target-volume(* 8 1024)
                            :spatial-topology "S2"
                            :boundary-conditions "PERIODIC")
```

  If you don't include these options and you use "OPEN" boundary conditions, then the boundaries will be minimal triangulations of the sphere: tetrahedra.

- The input to the "initial-spatial-geometry" and "final-spatial-geometry" options should be a list of space-separated lists, each containing three points (for a triangle, you see). Using this format, we can completely describe a 2D triangulation. For instance, a tetrahedron is

```
((4 3 2) (4 1 3) (1 4 2) (2 1 3))
```

  Give each point a number, and draw the triangles, and you'll see this is a tetrahedron. Generalize this format to input any boundary geometry homeomorphic to the sphere.n

Once you've initialized a spacetime, you also need to set the coupling constants. The two commands you can use for this are "set-k0-k3-alpha" and "set-k-litl-alpha." They both set all coupling constants, but in one case you directly set k and λ (called litL here) and in the other case you set k_0 and k_3, as discussed in Ambørn and Loll. In theory, in the fixed boundary conditions case, k_0 and k_3 should be radically different functions of k and λ than they are in the periodic boundary conditions case. However, so that we can easily compare k and λ between simulations, the formulae for k_0 and k_3 are identical to what they are in the fixed boundary conditions case.

$$\texttt{k} = k = \frac{1}{8\pi G}$$

and

$$\texttt{litL} = \lambda = k\Lambda.$$

k_0 =k0 and k_3=k3 are related to k and λ in a complicated way. α =alpha is the length-squared of time-like links. There is no benefit to making it anything but -1.

An example of the command syntax is

```
(set-k0-k3-alpha 1.0 0.75772 -1)
```

or

```
(set-k-litl-alpha 1 5.0 -1)
```

The values are in order. The first input is k. The second is litL. The third is alpha.

Once you've initialized the spacetime and set the coupling constants, you're ready to run simulations.

4.2 Loading an existing spacetime

To load a spacetime from a file, run:

```
(with-open-file (f "/path/to/filename.3sx2p1")
                (load-spacetime-from-file f))
```

The program will automatically build the spacetime defined in the file and set the coupling constants to the correct values.

5 Running a simulation

Once you've loaded or generated a spacetime, you might want to change a couple of parameters, by calling some commands like:

```
(setf *eps* 0.05)
(setf SAVE-EVERY-N-SWEEPS 500)
(setf NUM-SWEEPS 50000)
```

These all work the same as in the previous versions of the CDT code. The function "setf" changes a parameter as per common lisp standards.

- *eps*—The damping parameter. Changes the size of the critical surface.
- SAVE-EVERY-N-SWEEPS—How often you save a spacetime if you're generateing multiple in a single run.
- NUM-SWEEPS—The total number of sweeps the simulation goes through before finishing.

We also have a family of generate-* functions. These are the top-level functions that perform (sweep). Each collects data for a particular use-case. They are:

- **generate-data-console:** Periodically prints simplex counts and move acceptance ratios to the console. Good for tuning k0 and k3 parameters. Outdated with the addition of "map_phase_space_parallelized.py".
- **generate-data:** Periodically re-writes a single file using save-spacetime-to-file and a second file with the current progress.
- **generate-data-v2:** Like generate-data, but writes a new file each time so progress file is needed.
- **generate-data-v3:** Like generate-data-v2, but generates spatial 2-simplex and 3-simplex information every SAVE-EVERY-N-SWEEPS.
- **generate-movie-data:** Periodically appends to a file a new list of count-simplices-in-sandwich for every sandwich. Useful for visualizing the spacetime change over sweeps. Writes out a second file with progress data.
- **generate-spacetime-and-movie-data:** This is a combination of generate-data and generate-movie-data.

6 A complete script

This is an example of a complete script you could run to generate a new spacetime.

```
(load "cdt2p1.lisp")
(setf NUM-SWEEPS 50000)
(initialize-t-slices-with-v-volume :num-time-slices          28
                                   :target-volume            30850
                                   :spatial-topology         "s2"
                                   :boundary-conditions      "open"
        :initial-spatial-geometry "boundary_files/octahedron.boundary"
        :final-spatial-geometry   "boundary_files/octahedron.boundary")
(set-k0-k3-alpha 1.0 0.75772 -1)
(generate-spacetime-and-movie-data)
```

This is an example of a complete script you could run to load a spacetime from a file and generate an ensemble of 1000 spacetimes. (Note that there's a better way to do this if you have access to multiple cores. I'll get to that in a minute.)

```
(load "cdt2p1.lisp")
(setf NUM-SWEEPS 500000)
(setf SAVE-EVERY-N-SWEEPS 500)
(with-open-file (f "/path/to/file.3sx2p1")
                (load-spacetime-from-file f))
(generate-data-v2)
```

7 The fastest way to build an ensemble

The user would be wise to generate a new spacetime and thermalize it before generating an ensemble. We only want to add probable spacetimes to our path integral. Once a spacetime has been thermalized, however, we can take advantage of parallelization to speed up the process of building an ensemble. The algorithm goes something like this:

1. Generate a spacetime from scratch and let it thermalize for a long time (maybe 50000 sweeps or so).
2. Run a simulation that loads that spacetime, uses generate-data-v2 and saves every 500 sweeps or so. Run said simulation as many times as you have CPU cores on your computer.

I have written scripts to make this easier. The post-thermalization script is slihtly different than the load-spacetime-from-file listed above:

```
;;;; default_post_thermalization.script.lisp
;;;; Author: Jonah Miller (jonah.maxwell.miller@gmail.com)

;;;; Loads a thermalized file and calculates how many sweeps are
;;;; necessary given parallelization.

(load "cdt2p1")

;;;; Constants
(defvar *filename* "/path/to/file.3sx2p1"
  "The file we will use.")
(defvar *num-cores* 28 "The number of cores we can run the simulation on.")

(setf SAVE-EVERY-N-SWEEPS 500) ; How different each element of the
                               ; ensemble should be.
(defvar *ensemble-size* 1000 "How many spacetimes we want total.")

(defvar *total-sweeps* (* *ensemble-size* SAVE-EVERY-N-SWEEPS)
  "How many sweeps we need to get the ensemble we want.")
(defvar *sweeps-per-core* (ceiling (/ *total-sweeps* *num-cores*))
  "The number of sweeps used per core.")

(setf NUM-SWEEPS *sweeps-per-core*)

(with-open-file (f *filename*) (load-spacetime-from-file f))

(generate-data-v2)
```

You can find this exact file in /cdt_scripts/default_post_thermalization.script.lisp. Obviously, it needs to be changed depending on the number of cores the user has, the ensemble size required, etc.. Don't bother running the script *num-cores* times, though. Instead, use the shell script /start_cores.sh.

```
SCRIPTNAME="default_post_thermalization.script.lisp"
LOGFILENAME="ensemble_building.log"

for i in {1..28}; do
    nice sbcl --dynamic-space-size 2000 --script $SCRIPTNAME >> $LOGFILENAME &
    sleep 2s
    echo $i
done

echo "All_finished"
```

This script starts the script you put in SCRIPTNAME 28 times. Change the number 28 to the number of cores you have. Then just run ./start_cores.sh.

8 Additional Programs

There's a lot more to the CDT package than the simulations. There are a number of data analysis tools under data_analysis_scripts. Those files should be well documented in the comments—well enough to run them. Also check out the other files in the documentation folder.