2+1-dimensional Fixed Boundaries CDT: Programmer's Guide

Jonah Miller
jonah.maxwell.miller@gmail.com

August 27, 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 how to modify the code. I will first give a very brief overview of the algorithm of the program. Then dive into data structures and specific functions and how they should be used. This document assumes the reader knows how to run the program, its basic structure, and the physics behind it. In order, these can be found in:

- The user's guide.
- The file called program_and_module_list.txt
- Ambjørn and Loll's "Dynamically Triangulating Lorentzian Quantum Gravity," the documentation file "Gibbons-Hawkin_Boundary_Term_in_Causal_Dynamical_Triangulations.pdf," and my REU writeup, which is bundled here as well.

I also assume the reader knows some basics about programming in lisp. Here are some of the resources I used:

- Peter Siebel's "Practical Common Lisp" is an excellent, readable, and in-depth guide to writing code in lisp. It was my primary resource and you can find it online.

  `http://www.gigamonkeys.com/book/`

- The canonical work on Lisp is Paul Graham's "On Lisp." It is available online.

  `http://www.paulgraham.com/onlisp.html`

- Successful Lisp is another book available for free online.

  `http://psg.com/~dlamkins/sl/contents.html`

- This website links to a number of useful resources.

  `http://www.apl.jhu.edu/~hall/lisp.html`

Hopefully this guide will help elucidate how to edit the CDT code-base. If you have any questions, don't hesitate to email me.

2 The Algorithm

Although I'm sure the reader is familiar with the algorithm of CDT, I present it here for completeness. The metropolis algorithm as applied to CDT is as follows:

1. Generate an arbitrary spacetime constructed of equilateral simplices.
2. Randomly apply one of the ergotic moves to the spacetime.
3. Calculate whether the new spacetime is more or less likely than the old spacetime, or less, as given by its un-normalized weight in the partition function,[1] e^{iS}.
4. If the new spacetime is more likely than the previous spacetime, accept the change with probability $P = 1$. If it is less likely, accept the change with probability $P = P(new)/P(old)$. We can rewrite this probability as
$$P = e^{iS_{new} - iS_{old}}.$$
5. Return to step 2, and repeat until the spacetime is acceptably probable.

The primary data structures we need to worry about, then, are the geometric objects. These, and the functions that manipulate them, will be discussed in section 3. The most important functions are the action and the ergotic moves. These will be discussed in some detail in section 4.

3 Data Structures

The most important data structures in the simulation are, of course the goemetric objects that make up the spacetimes. These are points, edges, triangles, and tetrahedra.

The other important data structures in the simulation are the F and B vectors, which count geometric objects. We will discuss these after we discuss the geometric objects.

[1] Technically, the normalized weight $\frac{1}{Z}e^{iS}$ is the important quantity. However, since we only care about the ratio of weights, the factor of $1/Z$ cancels out. This is fortunate, since we don't know what Z is—if we did, we could simply perform the path integral.

3.1 Points

Points are represented as integer identifiers. There is no master list of points, but, whenever a new point is created, the value of a global variable is incremented to obtain the next point ID. Points have no coordinates in this program. In some sense, points are the most important objects in the simulation. As will be discussed below, the higher-dimensional simplices are partly defined by which points they contain. Objects that manipulate points are:

- ***LAST-USED-POINT***—This is the global variable that is incremented to obtain point ids. It starts at 0.
- **next-pt**—This is a function that increments *LAST-USED-POINT* by 1. It then returns the incremented *LAST-USED-POINT*. Call it with

```
(next-pt)
```

  Thus,

```
(format t "The next point is ~A~%." (next-pt))
```

  would return

```
The next point is <next point>.
```

- **set-last-used-pt**—In certain circumstances, for instance when loading a spacetime from file, we just want to tell the simulation how many points IDs have been used, rather than increment the global variable. The call

```
(set-last-used-pt point-number)
```

  will set *LAST-USED-POINT* to point-number.

LAST-USED-POINT and all objects that manipulate it can be found in "globals.lisp."

3.2 Links/Edges

Links are the edges of tetrahedra. There can be time-like links which connect points at two different times and space-like links which connect points at the same time. Since they are one dimensional, we also call links 1-simplices. Links, faces, and tetrahedra are stored in hash tables.[2]

Space-like links are stored in the hash table named

```
*SL1SIMPLEX->ID*
```

[2]A hash table is a generalized list. You can think of it as a function in the mathematical sense: It maps a *key* to a *value*.

and time-like links are stored in the hash table named

```
*TL1SIMPLEX->ID*
```

As the name implies, the key of the hash table contains the geometric object itself. The value associated with each key is actually just **0**. For the lower-dimensional simplices, only the keys are really used. Because hash tables can be thought of like mathematical functions, a given key can't be entered into the hash table twice—it just doesn't make sense. If I tried to add a new entry to the hash table that shared a key with an existing entry, the value of the existing entry would be overwritten, but otherwise nothing would change. By storing geometric information in the keys of a hash table, we ensure that we have no duplicate geometric objects.

3.2.1 *SL1SIMPLEX->ID*

The keys for the *SL1SIMPLEX->ID* hash table are of the form:

```
(tslice (p0 p1))
```

where tslice is the proper time the edge lives in, or the time slice that contains the edge, and p0 and p1 are the endpoints of the edge. The parentheses represent that the key is a list. (p0 p1) is another list. As discussed above, the value for each entry in the hash table is **0**.

4 Functions