Introduction to the LISP-based (2+1)-dimensional Causal Dynamical Triangulations code

David Kamensky

September 1, 2010

Abstract

The purpose of this document is to accelerate the reader's process of understanding of the Common LISP-based 2+1 dimensional Causal Dynamical Triangulations (CDT) code. I will describe the purpose, function, usage, and structure of the program at various levels of abstraction. This paper, and the code itself for that matter, assume that the reader has access to and interest in understanding the source code. There is no "user interface" in the traditional sense, just a collection of inter-related variables, functions, and macros for use in external LISP scripts. Most of what I present here applies to the 3+1 dimensional code as well.

1 Introduction

1.1 Purpose

1.1.1 CDT in a nutshell

I assume that the reader is familiar with the concepts of CDT, but, for completeness, I will briefly describe them here.

The theory revolves around computing quantum gravitational amplitudes by performing path integrals over spacetime geometries, using the Einstein-Hilbert action in the exponential weight of each geometry. It is separated from other types of dynamical triangulations by the fact that it only integrates over "causal" geometries: spacetimes for which there exist foliations into spacelike sheets of constant topology.

Most research into CDT revolves around numerical simulations in which spacetimes are represented approximately as simplicial manifolds. In the 2+1 case, a simplicial manifold is an arrangement of tetrahedra, all meeting at faces. The causality restriction adds the stipulation that tetrahedra must have at least one vertex on each of two space-like slices of triangles. In this simulation, the edge lengths of tetrahedra are held constant. The tetrahedra (simplices) are

internally-flat, but one can focus different amounts of curvature onto an edge by arranging more or fewer tetrahedra about that edge.

1.1.2 Examples of simulations

What would one do with a program that simulates CDT? The simplest question to ask is "what is the amplitude for transitioning from space g_a at time t_a to space g_b at time t_b?". With this program, the user can answer such a question, assuming that he or she can generate triangulations of g_a and g_b. The program can take the two triangulations as inputs and integrate over the interpolating causal spacetimes. The dedicated user could plug in many boundary geometries and come up with a wavefunction for the 2+1 universe.

Given that the user may also recover information about each spacetime that contributed to an integral, a user might examine the distributions of any number of geometrical observables and/or compare histories to the classical path.

1.2 Function

First, I will present a bird's-eye view of the algorithm used to integrate over spacetimes. It is a Monte Carlo algorithm that relies on a Wick rotation to substitute a partition function for the standard path integral and treat geometries as Euclidean triangulations. The algorithm consists of a random walk through geometries in which the probabilities of reaching different triangulations are forced, through a Metropolis algorithm, to match up with the distribution from the partition function.

The important mathematical result for understanding the code is that the ratio of the probabilities of visiting two different geometries is the exponential of the difference of the (Wick-rotated, Euclidean) geometries' actions: $P_a/P_b = e^{S_b - S_a}$. Thus the probability of making a certain move in the random walk (a small change to the geometry) depends only on the change in action due to that move. I have now given sufficient background to present the algorithm in pseudocode:

```
RandomWalk(T_0, N)
1  while (number of applied moves) < N
2        do
3           m ← (random candidate move)
4           if m is possible
5                 do
6                    ΔS ← (change in action due to m)
7                    r ← (random real number ∈ [0, 1.0])
8                    if (r < e^{-ΔS}), T_0 ← m(T_0)
```

The above obviously omits many technical details, not least the generation of the initial geometry, T_0. It conveys, however, the essentials of what the program does. The program contains several variations of this basic structure, each of

which outputs different geometrical data over the course of the random walk. That data may then be subjected to further analysis.

1.3 Usage

In this section, I will present several examples of user-created scripts and describe what they specify. This section is not an exhaustive user manual. It is intended to provide some basic insight into how one might use the code, along with some runnable examples. I will go into more detail about the exact meanings of the functions and their parameters in Section 2.

1.3.1 Small-volume debugging script

Running a serious integration requires at least several days on a modern workstation. When modifying the code, such integrations are impractical for testing and debugging incremental changes. The following is a simple script that, when executed from within the same directory as the source code, provides a quick check of integrity:

```
1  ;;load all of the simulation code
2  (load "cdt2p1.lisp")
3  ;;clear data from any previous runs
4  (reset-spacetime)
5  ;;set the number of sweeps (proportional to number of moves)
6  (setf NUM-SWEEPS 1000)
7  ;;generate an initial triangulation to start the random walk
8  (initialize-T-slices-with-V-volume :num-time-slices      8
9                                     :target-volume        8192
10                                    :spatial-topology     "s2"
11                                    :boundary-conditions "open"
12                                    :initial-spatial-geometry "tetra.txt"
13                                    :final-spatial-geometry   "tetra.txt")
14 ;;set the inverse Newton's constant and cosmological constant
15 (set-k-litL-alpha 0.20 5.00 -1.0)
16 ;;perform random walk
17 (generate-data-console)
```

The above, when run, will output some initial information and then repeatedly print sets of numbers to the screen. The initial information describes the starting point of the random walk and the sets of numbers describe the geometry as the random walk proceeds.

Note the value of the keyword arguments `initial-spatial-geometry` and `final-spatial-geometry`. These are filenames specifying from where the program should read geometry for the initial and final spatial slices. The file `tetra.txt` is a file containing the data

```
;;triangulation of a tetrahedra

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

This specifies a triangulation of a single tetrahedron, the smallest simplicial manifold of topology S^2 (topology of a 2-sphere). The integers identify points and the three-tuples identify triangular faces. The user could specify other filenames of files containing other triangulations of S^2 manifolds, in the same LISP-readable format.

1.3.2 Generating movie data

A helpful visualization of a random walk is a movie depicting a graph of the volume distribution per time slice, where each frame of the movie is a step in the random walk. The Lisp code can output the data necessary to generate such a movie. To create a movie, the user would run a script such as the following:

```
(load "cdt2p1.lisp")
(reset-spacetime)
(setf NUM-SWEEPS 100000)
(initialize-T-slices-with-V-volume :num-time-slices      64
                                   :target-volume        80000
                                   :spatial-topology     "s2"
                                   :boundary-conditions  "open"
                                   :initial-spatial-geometry "tetra.txt"
                                   :final-spatial-geometry   "tetra.txt")
(set-k-litL-alpha 0.20 5.00 -1.0)
(generate-spacetime-and-movie-data)
```

The above is similar to the quick test script, but uses a larger volume and calls `generate-spacetime-and-movie-data` rather than `generate-data-console`. This program will generate three automatically-named files in the working directory: one with the extension `.3sx2p1`, another with the extension `.mov2p1`, and a third ending in `.prg2p1`. The first is a snapshot of all spacetime geometry, overwritten at periodic intervals. The second accumulates lines specifying the volume per slice at periodic intervals. The third summarizes the progress of the simulation. The `.3sx2p1` file may be used to restart the simulation if the process dies prematurely.

2 Code structure

2.1 Major data structures

2.1.1 Abstract description of hash tables

To understand the code, the reader must have an understanding of hash tables. For those not familiar, I will present a brief summary of the expected behavior of a hash table. A hash table is a collection data structure that stores (key, value) pairs. One may usually access, add, or delete a value using its key in constant time (although rare, worst-case scenarios may cause access time to be linear in the size of the table). The key need not be an integer, but don't hesitate to think of it like an index into an array storing the values. Lisp has built-in support for hash tables.

2.1.2 Representation of geometry

The defining aspect of this program's design is its representation of spacetime geometry. The program contains explicit representations of points, faces, and tetrahedra. These representations are described below:

- **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.

- **Faces:** Faces (triangles/2-simplices) are stored as values in a hash table with integral keys. The triangles are represented as Lisp lists, containing the point IDs of the corners, the number of points on the lower time slice (referred to as the "type") and the upper and lower time values. The data from a triangle may be accessed through the following API of clearly-named macros:

```
(defmacro 2sx-type (2sx) `(first ,2sx))
(defmacro 2sx-tmlo (2sx) `(second ,2sx))
(defmacro 2sx-tmhi (2sx) `(third ,2sx))
(defmacro 2sx-points (2sx) `(fourth ,2sx))
```

  The program uses a second hash table to store triangle IDs, keyed by triangle data. The purpose of this is to prevent geometrically-equivalent triangles from being aliased by multiple IDs. If a triangle is being created during a move on the geometry, the program checks the new triangle data against this second hash table, creating a new entry in the first hash table only if the desired triangle does not already exist.

- **Tetrahedra:** Tetrahedra (3-simplices) are stored in a way similar to triangles. They are also lists keyed by integers in a hash table, but, in addition to type, low time, high time, and points, 3-simplex lists contain lists of the IDs of their faces and neighboring 3-simplices. This data is all accessible through the following API:

```
(defmacro 3sx-type (sx) `(nth 0 ,sx))
(defmacro 3sx-tmlo (sx) `(nth 1 ,sx))
(defmacro 3sx-tmhi (sx) `(nth 2 ,sx))
(defmacro 3sx-points (sx) `(nth 3 ,sx))
(defmacro 3sx-sx3ids (sx) `(nth 4 ,sx))
(defmacro 3sx-sx2ids (sx) `(nth 5 ,sx))
(defmacro 3sx-lopts (sx) `(subseq (3sx-points ,sx) 0 (3sx-type ,sx)))
(defmacro 3sx-hipts (sx) `(subseq (3sx-points ,sx) (3sx-type ,sx)))
```

  Keep in mind that the relative orderings of point, triangle, and 3-simplex IDs is important in the lists retrieved by the above functions. Face and tetrahedron IDs are stored at the same indices as the points opposite them.

  Tetrahedra require no additional reverse hash table because no procedures need to check for duplicate 3-simplices.

2.2 A more detailed description of the Monte Carlo algorithm

The pseudocode in 1.2 ignores the actual code's grouping of moves into "sweeps" and glazes over how moves are chosen, represented, and applied. The data generation functions (`generate-data-console`, `generate-movie-data`, etc.) do not directly perform the loop given by the pseudocode. Instead, they call the function `sweep` a certain number of times, outputting data between sweeps. A sweep consists of the loop from the pseudocode, with N equal to the number of 3-simplices in the target volume. This provides a clear way to compare how long simulations of different volumes have been running; the total number of attempted moves scales with volume for a constant number of sweeps. The code for the function `sweep` is given below:

```
(defun sweep ()
  (let ((num-attempted 0))
    (while (< num-attempted N-INIT)
      (let* ((sxid (random *LAST-USED-3SXID*))
             (mtype (select-move))
             (movedata (try-move sxid mtype)))
        (while (null movedata)
          (setf sxid (random *LAST-USED-3SXID*)
                mtype (select-move)
                movedata (try-move sxid mtype)))
        (incf num-attempted)
        (incf (nth mtype ATTEMPTED-MOVES))
        (when (accept-move? mtype sxid)
          (incf (nth mtype SUCCESSFUL-MOVES))
          (2plus1move movedata))))))
```

In words, this function does the following: while the number of possible moves found is less than the target volume, try selecting random move types and simplices until a possible move is found and use the Metropolis algorithm (hidden behind `accept-move?`) to randomly decide whether to apply the move.

A deeper look at exactly what each function does may be informative. The function `try-move` takes a 3-simplex ID and a move type, then checks whether the simplices around the identified tetrahedron will permit the requested move type. If the move is possible, `try-move` plays a "what if" game, generating a list, `movedata`, that details what geometry would be created and/or deleted by applying the move. If the move is impossible, `movedata` is `nil`. The move type and simplex ID are sufficient for `accept-move?` to determine the would-be change in action of the manifold and decide whether or not the move should be applied. If `accept-move?` gives the go-ahead, `2plus1move` acts on the description of changes given by `movedata`, goes over to the hash tables, modifies geometry, and updates some counters.

2.3 Descriptions of individual file contents

In the following subsubsections, I will elaborate on some implementation-level details of each file loaded by `cdt2p1.lisp`.

2.3.1 File `globals.lisp`

As its name suggests, this file contains definitions of the global parameters and a few functions used by routines in the other files. The most prominent global variables are the hash-tables, the f-vector and the b-vector. I described the hash tables in 2.1. The f-vector is a collection of variables counting the numbers of different geometrical entities. These variables are updated every time a move is made. The b-vector is an analogous set of variables pertaining to the boundary geometry. An important convention to note is that the f-vector counts geometrical entities indiscriminately; to count up geometry in the bulk only, the programmer must consider both the f- and b-vectors; this does not entail merely subtracting them; think about the semantics of the vectors before invoking them.

The most noteworthy function in `globals.lisp` is `action`. This computes the action of a simplicial manifold described by its arguments. The action function is merely a linear combination of several integer arguments counting the numbers of geometric entities (as per Regge calculus).

2.3.2 File `simplex.lisp`

This file defines how 2- and 3-simplices are represented, created, stored, accessed, connected, counted, and serialized. I discussed the representation and member accessor functions for 2- and 3-simplices in 2.1.2.

There are several functions to create new simplices, each convenient in different scenarios. These functions are named in the form `make-*simplices*` and vary with respect to how simplex data is represented by the arguments. These functions all modify the relevant hash tables, but some don't "connect" new 3-simplices to the surrounding geometry; the IDs of neighboring faces and tetrahedra are left as zero.

The function `connect-3simplices` performs this task. The other `connect-*` functions may be used to apply `connect-3simplices` to larger groups of tetrahedra.

The `get-*` and `count-*` functions use LISP's `maphash` function to filter the hash tables for geometry meeting certain conditions, returning either a list of the relevant geometry or a count of it. These are expensive in time because they must access every element in the hash table; a better way to count geometry in the middle of the simulation is to look at the f-vector (see 2.3.1).

2.3.3 File `moves.lisp`

The functions in this file are used to facilitate the application of moves to the geometry. The first function is the one that actually applies moves: `2plus1move`. It takes in a list of information, `movedata`, that describes the move to be made in terms of what simplices are to be added, removed, and modified.

The remaining functions are used to probe for possible moves and return descriptions of those moves that may then be passed to `2plus1move`. Functions with names of the form `*-subcomplex` take a simplex ID and try to find a group of neighboring simplices appropriate for a certain move. Functions with names of the form `try-*` take a simplex ID, use the associated subcomplex function to grab some neighbors of that simplex, then prepare a description of the would-be move. Both the try and subcomplex functions return `nil` if the requested move/ID combination is impossible.

2.3.4 File `initialization.lisp`

This file contains the functions used to generate the starting point of the random walk (T_0 from the pseudocode of 1.2). It is home to the initialization function called by the example scripts of 1.3. The number of "error: not implemented" messages that this code may generate should serve as a clue about its status; it is unfinished and subject to change. That said, some of the approaches taken by it will most likely remain the same in future versions. The most general of these is the method of starting with a low-volume spacetime and repeatedly applying growth moves to it until the desired target volume is reached. As an REU student, I added the functionality to load arbitrary boundary geometries

for the first and last time slices; so long as that bit of code continues to be used, I refer the reader to my REU paper for a description of how it works.

2.3.5 File `montecarlo.lisp`

This is where the core of the Monte Carlo algorithm resides. The data generation functions, `sweep`, and `accept-move?` are within. I described these functions at several levels of intricacy before, so I direct the reader back to 2.2 for a detailed discussion.

3 Conclusion

I have left many details unaccounted for. Describing the code further, however, would distract from its salient features. At a certain point, reading the source code is the best way to learn the program. It is written in Lisp and sacrifices performance for the sake of understandability; with a little background (such as reading this document), a programmer should have no trouble extracting meaning directly from the code. The most significant gap in the coverage of this document is its lack of description of the different geometrical moves. A thorough description of these already exists in the paper *Dynamically triangulating Lorentzian quantum gravity* (Nuclear Physics B 610 (2001) 347-382), by J. Ambjørn et al. – it is essential reading for anyone working with this program.