Infrastructure for Simulating n-Dimensional Simplicial Complexes

Dušan Cvijetić, Nenad Korolija, and Marko Vojinović

***Abstract*—We present an infrastructure for simulating simplicial complexes. The classes for storing the structure of simplicial complexes and simplices are explained in detail, for arbitrary dimension.**

The implementation is tested using functions for seeding simplicial complexes and for printing them on the screen. Beside these functions, the supporting classes and the function for assigning unique identifiers and screen coordinates is also explained.

Results of simulation show that there are potentials for the simulator to be used for big data problems, although appropriate experimental results are still being collected. Future work includes parallelizing the execution of the simulator using supercomputing architectures.

***Index Terms*— Simplicial complex; triangulation; manifold; algebraic topology.**

I. Introduction

A manifold is one of the fundamental concepts in mathematics [1], and its importance in applications in physics, technology and engineering cannot be overstated. Virtually all modern physics describes the world using *field theory* [2], in which all physical quantities (fields) are represented as functions over some manifold (for example, spacetime). In technology, manifolds appear in all forms and guises, whenever one needs to deal with curved surfaces --- from civil engineering to graphics in video games.

While most of the interest in science and engineering revolves around *smooth* manifolds, for the purpose of studying manifolds using numerical techniques, the attention focuses on the so called *piecewise-linear* manifolds [3], which can intuitively be imagined as a structure made out of small flat cells called *simplices*, arranged like bricks into a structure which models a manifold. The procedure of approximating a smooth manifold with a piecewise-linear one is commonly called *triangulation*, see Fig. 1.

Within the framework of algebraic topology, the formal mathematical structure which describes piecewise-linear manifolds is called a *simplicial complex*. For the purpose of this article, we provide an informal descriptive definition of a simplicial complex, without mathematical rigour. A simplicial complex is a combinatorial structure, containing the information about *simplices* of various dimensions that make up a complex, and the information about how simplices are connected to each other. A *k-simplex* is an elementary building block of a simplicial complex. It is an elementary geometrical "cell" of dimension k, which is being used to build simplices of higher dimension, and the entire simplicial complex. For k = 0, the simplex is called a *vertex*, it is represented geometrically as a single point, and has no internal structure. The k = 1 simplex is called an *edge*, geometrically represented as a single straight line, having two vertices at its boundary. For k = 2, the simplex is a *triangle*, having three boundary edges and three vertices. The case k = 3 describes a *tetrahedron*, having four boundary triangles, six edges and four vertices. One can go further into higher dimensions: k = 4 represents a simplex called *pentachoron* – it is a 4-dimensional figure, having five boundary tetrahedra, 10 triangles, 10 edges and five vertices. In general, one can introduce a k-simplex for arbitrary dimension k, also called *level*.

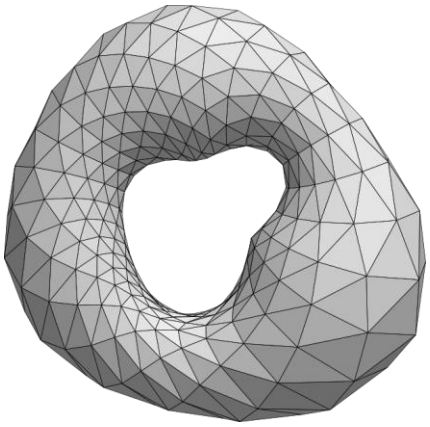


Fig. 1. Simplicial complex of a torus (source: Wikipedia).

Given a set of simplices, one can "glue them up" into a bigger geometrical structure, called simplicial complex. In order to describe a manifold of dimension D, a simplicial complex is constructed by gluing a set of D-simplices by identifying their common boundary (D-1)-simplices. Naturally, this implies the identification of all corresponding sub-simplices of level k < D-1 as well. The resulting simplicial complex is homeomorphic to a piecewise-linear manifold of dimension D.

The most important information about the simplicial complex, aside from its dimension D, is the data that tells one

Dušan Cvijetić is a student of the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: dusancvijetic2000 @ gmail.com).

Nenad Korolija is with the School of Electrical Engineering, University of Belgrade, 73 Bulevar kralja Aleksandra, 11020 Belgrade, Serbia (e-mail: nenadko @ etf.bg.ac.rs).

Marko Vojinović is with the Institute of Physics, University of Belgrade, Pregrevica 118, 11080 Pregrevica, Serbia (e-mail: vmarko @ ipb.ac.rs).

 ISBN 978-86-7466-930-3

which simplices are glued to which. This gives rise to a notion of a *neighborhood* of a k-simplex, which is a set of all simplices which contain a given simplex as its sub-simplex (called super-neighbors) and simplices which are contained in a given simplex (called sub-neighbors). Each k-simplex (for $0 \leq k \leq D$) in the complex has its set of neighbors, where by definition a simplex is not a neighbor of itself (this is convenient to avoid infinite loops when traversing a complex). The neighborhood structure of the entire complex determines the *topology* of the corresponding manifold.

While manifolds of various topologies are important in their own right in mathematics, the applications in physics and engineering typically introduce functions over manifolds, such as distances, areas and volumes, temperature, electric and magnetic fields, etc. In the language of simplicial complexes, these functions are commonly called *colors*, and are assigned to simplices of various level k within the complex. Given a k-simplex, one can assign to it multiple colors, representing the value of a given function when evaluated on the k-simplex. A prototype example of colors is the geometry of a simplicial complex: each k-simplex is assigned its "size" according to its geometry --- each 1-simplex (an edge) is assigned a real number representing its length, each 2-simplex (a triangle) is assigned a real number representing its area, tetrahedra are assigned volumes, and so on. Other examples are abound --- vertices can be assigned a temperature, edges can be assigned vectors of electric field, and so on. Depending on the problem at hand, one may or may not impose relationships between various colors, such as that the area of a triangle is consistent with the length of its edges, or similar. These relationships are collectively called *constraints*.

In most everyday applications, one is interested in manifolds of dimension 1 and 2 (curves and surfaces). However, within the context of theoretical physics, one often needs to deal with manifolds of higher dimension – most commonly 3, 4, 5, 10, 11 and 26, while more sporadically anything in between and above. One of the typical scenarios is *quantum gravity* [4,5], a vast research area of fundamental theoretical physics, where the notion of spacetime is described as a piecewise-linear manifold of dimension D=4 or higher [6,7]. In order to apply numerical techniques to study the manifolds in such research disciplines, it is necessary to formulate and implement structures and algorithms which describe colored simplicial complexes of arbitrarily large dimension, in a uniform and optimal way. In what follows, we describe one such implementation, which is purposefully designed to mimic the mathematical structure of a simplicial complex as close as possible, while simultaneously providing efficient numerical techniques for the manipulation and study of such structures.

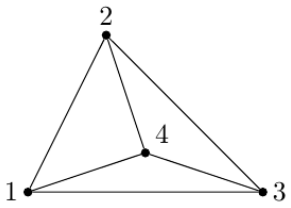
I. N-Dimensional Simplicial Complexes

This section describes the structure of simplicial complexes, and explains an example C++ implementation of classes for storing simplicial complexes.

Simplicial complexes consist of k-simplices at different levels. Given a simplicial complexes of dimension D, these elements include k-simplices for each level from zero to D. Elements at level zero are vertices, elements at level one are edges, elements on level two are triangles, etc. Finally, there are elements of highest level D. The representative source code of class for simplicial complexes is given in Algorithm 3 from the Appendix. The source code is pruned from comments and unnecessary functionalities for the presentation of the simulator.

K-simplex stores the level it has, the dimension of the simplicial complex it belongs to, neighboring elements and colors assigned to it.

Neighboring elements of a k-simplex are defined as k-simplices that this k-simplex is touching. Since these can be on various levels, the structure of neighbors is the same as for the simplicial complex. Therefore, the two main classes are mutually connected.




```
Printing SimpComp tetrahedron, D = 3
Simplices k = 0:
1, 2, 3, 4
Simplices k = 1:
(1-2), (1-3), (1-4), (2-4), (2-3), (3-4)
Simplices k = 2:
(1-2-3), (1-3-4), (1-2-4), (2-3-4)
Simplices k = 3:
(1-2-3-4)
```

Fig. 2. Tetrahedron and a corresponding output of the simplicial complexes simulator.

One possible implementation of the neighboring elements is to store only neighbors from one level above, and one level beneath (first sub-neighbors and first super-neighbors). The lower- and higher-level neighbors can be deduced following the structure of the first neighbors. However, we have opted for storing neighbors from all levels, giving us the opportunity to divide the structure onto multiple computing nodes and run the code in parallel. At current state, the simulator is running on a single CPU.

The instructions a CPU is executing are repeated over and over again, which makes this simulator suitable for acceleration using the dataflow paradigm [8,9]. The effort required for programming such architectures is higher than for conventional von Neumann architectures [10], but the simulator is suitable for transforming the C++ source code automatically [11]. Executing multiple simplicial complex operations in parallel requires appropriate scheduling

 ISBN 978-86-7466-930-3

techniques [12].

Each k-simplex (including all vertices, edges, triangles, etc.) can be colored with different types of color. Example colors include:

- k-simplex name,
- unique identifier of k-simplex,
- boundary color of k-simplex,
- screen coordinates.

These colors are included in our simplicial complex simulator, but the structure of the simulator allows adding additional user defined colors.

The representative source code of the class for k-simplices is given in Algorithm 4 from the Appendix. Just like it is the case with simplicial complexes, this source code is pruned for better clarity.

For simulation purposes, we have developed functions for seeding simplicial complexes at various levels, as it will be explained in the following section. In addition, coloring and printing simplicial complexes is also implemented. Pretty printing (or compact printing) prints k-simplices at all levels, where k-simplices of level higher than zero are printed as tuples consisting of unique identifiers (IDs) of their vertices. Fig. 2 shows an example tetrahedron (i.e. simplicial complex of dimension D = 3 consisting of a single 3-simplex and its sub-simplices) whose vertices are colored with unique identifiers that auto-increment after each assignment of the unique color to a vertex. Details of the implementation of compact printing is also explained in this manuscript.

Screen coordinates can be attached to vertices of the tetrahedron. Therefore, it can be drawn on the screen. However, there is no need to assign coordinates. They are just a convenient way to show an object on a screen. Similarly, there is no need to assign unique ID to any vertex. In the previous example, if a vertex with unique ID four would not have a unique ID assigned to it, the tetrahedron could still be printed out, but with word “Simplex” being printed out in place of number four.

II. Seeding Simplicial Complexes

This section describes seeding simplicial complexes using C++ implementation of function *seed_single_edge()*. The example source code for seeding a single edge is used for demonstrating purposes.

The process of seeding simplicial complexes will be explained using the source code shown in Algorithm 1. The source code is pruned from comments and unnecessary statements. Seeding a simplicial complex consists of the following steps, and statements in Algorithm 1 follow the same principle in the same order:

- creating an empty simplicial complex of given dimension,
- creating k-simplices for storing vertices and simplices of higher levels,
- connecting vertices at each level with vertices on higher and lower levels.

Adding a neighbor to a k-simplex is a symmetric operation. This means that both k-simplices (the calling one and the one given as an argument) are neighbors to each other. All functions of the simulator are written in a robust manner, checking the validity of input parameters.

Note that multiple colors can be assigned to each k-simplex, which is left out of consideration in this algorithm for better clarity.

III. Coloring and Pretty Printing K-simplices

This section describes coloring and pretty printing simplicial complexes. These functions might work in pair, but are not necessarily connected.

A. Coloring K-simplices

Coloring k-simplices will be explained using Algorithm 2 by coloring vertices of an edge with boundary colors. First, vertices have to be created as k-simplices of level zero. Then, colors have to be created for all vertices. Finally, colors need to be pushed back to the vector of colors that each k-simplex has.

Algorithm 1: Seeding a single edge.

```
SimpComp* seed_single_edge(string name){
    SimpComp *edge = new SimpComp(
            name, 1);
    KSimplex *v1 =
            edge->create_ksimplex(0);
    KSimplex *v2 =
            edge->create_ksimplex(0);
    KSimplex *e1 =
            edge->create_ksimplex(1);
    v1->add_neighbor(e1);
    v2->add_neighbor(e1);
    return edge;
}
```

Algorithm 2: Coloring vertices with boundary color.

```
KSimplex *v1 =
        edge->create_ksimplex(0);
KSimplex *v2 =
        edge->create_ksimplex(0);
Color *c1 = new BoundaryColor(true);
Color *c2 = new BoundaryColor(true);
v1->colors.push_back(c1);
v2->colors.push_back(c2);
```

Following colors are currently available:

- unique ID colors
- boundary colors
- screen coordinate colors.

Additionally, user is allowed to construct a custom color and use it within the simulator. The source code of the simulator is organized as a library, and user is allowed to extend it by using the library.

Unique ID colors are predominantly used for pretty printing simplicial complexes. They are implemented by a class inherited from the basic color class. Two main fields include

 ISBN 978-86-7466-930-3

static integer number, and an integer number. The first represents the current maximum of a unique color ID that is in use, and the second one is the color of a given k-simplex.

Unlike unique ID colors, boundary colors have special meaning. Each k-simplex may contain boundary color, but it does not have to. A simplicial complex can have boundaries on k-simplices of one level lower than the dimension of the simplicial complex. For example, a triangle can have edges as boundaries.

Screen coordinate colors are used for drawing simplicial complexes on a screen. The basic graphical user interface is under development.

B. Pretty Printing K-simplices

Printing k-simplices includes printing of all of the fields that *KSimplex* class contains. This includes printing all of the neighborhood elements the k-simplex has. This is usually overwhelming for a user. Therefore, pretty printing is designed to print unique ID colors of each k-simplex in most readable way authors could think of.

Function *KSimplex::print_compact()* is responsible for pretty printing. It assigns to the pointer to the unique ID a value returned by a function *get_uniqueID()* that returns either nullptr if a k-simplex doesn't have a unique ID, or a pointer to the color.

If there is no unique ID color assigned to a k-simplex, the output consists solely of word "Simplex". Otherwise, *print_compact()* function is called for a color that the pointer points to. Further, the following procedure is repeated, if level k is greater than zero and there are neighboring elements for all neighbors. A set of integer values is constructed, and then function *print_vertices_in_parentheses(s)* is called for neighbors, adding unique IDs to the set. This way, printing sorted values is achieved, along with avoiding duplicate values. Sample output of a simplicial complex pretty printing is shown in Fig. 1.

IV. Conclusion

We have demonstrated how one can implement in code the structure of a simplicial complex of arbitrary dimension, in a way that is faithful to its combinatorial definition, and perform the most basic operations on it, like instantiating, coloring and printing.

The implementation of the basic classes of the code described in this work represents a fundamental basic building block for a more versatile software collection that aims to construct, manipulate and study the properties of simplicial complexes of arbitrary dimension. Future extensions of the software library will include the functions which implement attaching additional simplices to a boundary of a complex, performing Pachner moves [13] which transform a given complex into a different one without changing its topology, and functions for manipulating the colors and evaluating various mathematical constructions that include them. Note that the experimental data regarding the parallelization is yet to be collected (see the accompanying paper [14]).

The resulting software collection will feature the generality and versatility that aim for applications both in pure mathematics (algebraic topology research) and theoretical physics (quantum gravity, field theory), but also with potential applications in other disciplines of engineering and industry, wherever the analysis and the study of geometry of manifolds and curved surfaces may be relevant.

Appendix

Algorithm 3: Declaration of SimpComp class.

```
class SimpComp{
public:
    SimpComp(int dim);
    SimpComp(string s, int dim);
    ~SimpComp();
    int count_number_of_simplexes(
            int level);
    void print(string space = "");
    bool all_uniqueID(int level);
    void collect_vertices(set<int> &s);
    void print_set(set<int> &s);
    void print_vertices_in_parentheses(
            set<int> &s);
    void print_compact();
    // Creating new KSimplex at level k:
    KSimplex* create_ksimplex(int k);
    void print_sizes();

    string name;
    int D;
    // An element at each level
    // is a list or vector
    // of KSimplex pointers
    // to KSimplex on that level:
    vector< vector<KSimplex *> >
            elements;
};
```

Algorithm 4: Declaration of KSimplex class.

```
class KSimplex{
public:
    KSimplex();
    KSimplex(int k, int D);
    ~KSimplex();
    bool find_neighbor(KSimplex *k1);
    void add_neighbor(KSimplex *k1);
    void print(string space = "");
    UniqueIDColor* get_uniqueID();
    void print_compact();

    int k; // level
    int D; // dimension
    SimpComp *neighbors;
    vector<Color *> colors;
};
```

Acknowledgment

DC and NK were partially supported by the School of

 ISBN 978-86-7466-930-3

Electrical Engineering, University of Belgrade, Serbia. NK was partially supported by the Institute of Physics Belgrade, contract no. 0801-1264/1. MV was supported by the Science Fund of the Republic of Serbia, grant no. 7745968, "Quantum gravity from higher gauge theory" – QGHG-2021. All authors were partially supported by the Ministry of Education, Science, and Technological Development of the Republic of Serbia.

REFERENCES

[1] M. W. Hirsch, *Differential Topology*, New York, USA: Springer Verlag, 1976.

[2] A. Hobson, "There are no particles, there are only fields", *Amer. Jour. Phys.* **81**, 211-223 (2013).

[3] E. H. Spanier, *Algebraic Topology*, New York, USA: Springer Verlag, 1966.

[4] C. Rovelli, *Quantum Gravity*, Cambridge, UK: Cambridge University Press, 2004.

[5] C. Rovelli and F. Vidotto, *Covariant Loop Quantum Gravity*, Cambridge, UK: Cambridge University Press, 2014.

[6] T. Radenković and M. Vojinović, "Higher Gauge Theories Based on 3-Groups", *JHEP* **10**, 222 (2019).

[7] A. Miković and M. Vojinović, "Standard Model and 4-Groups", *Europhys. Lett.* **133**, 61001 (2021).

[8] B. Lee and A. R. Hurson, "Issues in dataflow computing," *Advances in computers*, Elsevier, **37**, 285-333 (1993).

[9] V. Milutinovic, J. Salom, D. Veljovic, N. Korolija, D. Markovic, and L. Petrovic, "Transforming applications from the control flow to the dataflow paradigm," *Dataflow supercomputing essentials*, Springer, Cham, 107-129 (2017).

[10] J. Popovic, D. Bojic, and N. Korolija, "Analysis of task effort estimation accuracy based on use case point size," *IET Software*, **9(6)**, 166-173 (2015).

[11] N. Korolija, J. Popović, M. Cvetanović, and M. Bojović, "Dataflow-based parallelization of control-flow algorithms," *Advances in computers*, Elsevier, **104**, 73-124 (2017).

[12] N. Korolija, D. Bojić, A. R. Hurson, and V. Milutinovic, "A runtime job scheduling algorithm for cluster architectures with dataflow accelerators," *Advances in computers*, Elsevier, **126** (2022).

[13] U. Pachner, "PL homeomorphic manifolds are equivalent by elementary shellings", *Eur. Jour. Combinat.* **12**, 129-145 (1991).

[14] D. Cvijetić, N. Korolija and M. Vojinović, "Possibilities for Parallelizing Simplicial Complexes Simulation", IcETRAN 2022, Novi Pazar, Republic of Serbia, June 6-9, 2022, Belgrade: Društvo za ETRAN, Beograd: Akademska misao (2022).

 ISBN 978-86-7466-930-3