Chain Models in Computer Simulation

R. Egli [a] N. F. Stewart [b]

[a] *Département d'informatique, Université de Sherbrooke, 2500 boul. Université, Sherbrooke, Québec, Canada J1K 2R1.*

[b] *Département d'informatique et de recherche opérationnelle, Université de Montréal, C.P. 6128, Succ. CentreVille, Montréal, Québec, Canada H3C 3J7.*

Abstract

This paper presents an extension of a previously-described framework for the specification and manipulation of models of systems. The original framework and its extension include a convenient Application Programming Interface (API), and this paper describes examples of the use of this API in the context of graphical simulation. These examples include the simulation of objects whose definition involves torques, and a brief description of the simulation of air and smoke by means of a model involving particles moving in fluid.

Key words: chain model, graphical simulation, cellular complexes, topological data structures

1 Introduction

As modeled processes and phenomena become more complicated, mechanisms for their concise, precise, and convenient description become more important. To illustrate, a computer-graphics program involving the simulation of fluid might involve hundreds of lines of code in a high-level language [9], and in such cases, a description language capable of specifying the simulation in a few lines provides a significant advantage. The method of description described here is appropriate for models in several areas, and it can be applied both to

⋆ The authors are grateful to V. Shapiro for many helpful comments. Responsibility for any errors lies entirely with the authors. The research was supported in part by grants from the Natural Sciences and Engineering Research Council of Canada.

Email addresses: Richard.Egli@USherbrooke.ca (R. Egli), stewart@iro.umontreal.ca (N. F. Stewart).

real physical simulations (see [7] and the related work in [4,12]) and to non-physical processes such as mesh-refinement.

The description mechanism extends work of Palmer and Shapiro [13,14], as well as [7]: it is a framework for the specification and manipulation of *attributes* associated with topological elements of objects. The word *object* refers to something more general than a model for a physical object, and as the mesh-refinement example illustrates, the method will apply even beyond physical systems. The word *framework* includes both a theoretical framework (a formalism) and a practical software kernel with a convenient API (Application Programming Interface).

The formalism is a *chain-model formalism* that permits the inclusion of arbitrary object-related data, including topological, geometric and physical data. Objects defined using the formalism are *abstract cellular complexes* which, like abstract simplicial complexes [10, p. 213], do not have geometric information associated with them *a priori*. The adjective "abstract" will be deleted for brevity, but we emphasize that in this paper, as in [5,7,8], "cellular complex" does not mean "geometric cellular complex" unless "geometric" is explicitly specified. This is in contrast to the common usage in the solid-modeling literature [2,17,18] (see also [11]). As in [7, Sec. 3], whether adjacency relationships are consistent, or make sense in the context of the specification of a geometric object, is considered to be the responsibility of the user of the API.

A cellular complex describes an object in terms of *cells* of various dimension, and specifies the topology of the object. Taking a geometric cellular complex as our first example [7], a cube can be decomposed into a 3-cell (the entire cube), six 2-cells (the faces, including the edges), twelve 1-cells (the edges, including the vertices), and eight 0-cells (the vertices at the corners of the cube). Similarly, two cubes sharing a face would have two 3-cells, eleven 2-cells, twenty 1-cells, and twelve 0-cells. A *chain model* permits the storage and manipulation, of information associated with various cells of the complex, by means of *chains* [5,7,13,14,17]. A *p-chain* is defined by associating a coefficient with each p-dimensional cell of the complex. These coefficients correspond exactly to the attributes mentioned earlier. We then define operations on chains, in order to permit global (over the entire object) manipulation of the coefficients. Examples were given in [7]: a mass-spring network modeling a flag [15], and classical Catmull-Clark subdivision [3] (see also [16]). In addition, however, we introduce in this paper the idea of associating a coefficient (attribute) with a *pair* of cells, along with the related idea of a chain of subchains, and we give concrete examples to illustrate the usefulness of these ideas.

It is not an accident that *topological* data structures are convenient for the description of processes and phenomena. Indeed, algebraic topology has been identified [4] as "... that common structure responsible for the apparent analo-

gies in physical theories ... [19,20]." It was shown in particular [4] that the ubiquitous balance laws of physics are fundamentally topological in nature, and also that "... most physical models may be classified based on their algebraic topological structure ... [21]." Furthermore, in many applications it is important to use data structures that are topological, but not *necessarily* geometric. For example, the *starplex* introduced in [4], for the representation of differential forms, is (like the *star* of algebraic topology [10, p. 204]), an abstract cellular complex that is not by itself geometric. Similarly, the example presented in [7, Sec. 4.1.2] involves a complex that is not wholly geometric. For some applications, of course, geometric complexes will be sufficient: an example is the use, in solid modeling, of modulo-2 homology [17].

In the next section we give a brief summary of the framework introduced in [7]. Then, in Section 3, the main theoretical contribution of this paper, the extension of the basic framework to permit *chains of subchains*, is described. We also introduce new operations, on these new entities, that will have direct and useful application, as well as a natural definition of duality for chains of subchains. In Section 4, we describe applications of the extended framework to computer graphics and other areas. Section 5 is the conclusion.

2 The basic framework

The basic topological elements in the formalism are 0-cells, 1-cells (which are collections of 0-cells), 2-cells (collections of 1-cells), and so on. The definition is recursive: a *p-cell* is composed of $p+1$ or more $(p-1)$-cells, $p \geq 1$, where the 0-cells (vertices) are taken as primitive. The set of p-cells is denoted by S_p, $0 \leq p \leq n$, and an *n-complex* K is defined as a collection of $n+1$ finite sets $S_0, S_1, \ldots, S_n$ together with a corresponding set of $n+1$ algebraic boundary operators ∂_p, $0 \leq p \leq n$; the operator ∂_p associates to each element of S_p some combination of elements of S_{p-1}, which constitute [1] its boundary. The cells in the boundary of a given cell are called the *faces* of the cell, and the cells whose boundary includes the given cell are called the *cofaces* of the cell; the face-coface relations capture all incidence information in a cell complex [7,14].

The construction and manipulation of entities such as cells and cellular complexes is implemented [1] in an API which includes the class *Complex*; this class has several methods associated with it, including a constructor, a method *dim* to return the largest dimension of a cell in the complex, as well as

- *int nb_cells(int p)*, which returns the number of cells of dimension p;
- *Cell cell(int i, int p),* which returns the i-th cell of dimension p;

[1] The complete code is available [6].

- *void add(Cell cellp),* which adds the p-cell identified by *cellp* to the complex (so that *K.cell(K.nb_cells(p),p)* would return the added cell as its value).

Notice that the order of the p-cells $C_i^p = K.cell(i,p)$, as a function of i, depends on the order in which the cells were added to the complex.

The API also includes the class *Cell*, which again has several methods associated with it: a constructor (which declares the cell), a method *dim* to return the dimension of the cell, and (amongst others) the methods

- *int nb_adj(int delta_p),* which returns the number of adjacent cells of dimension $p + \delta p$, where δp is specified by *delta_p*;
- *Cell adj(int i, int delta_p),* which returns the i-th neighbor (of class *Cell*) of dimension $p + \delta p$, δp is specified by *delta_p*.

Here, p is the dimension of the *Cell* instance invoking the method. The method *adj* generalizes the concepts of *face* and *coface*. The order in which it returns cells depends *not* on the order in which cells were constructed and added to the complex, as was the case for *cell(i,p)*, but, rather, on the order in which the cells were assembled using lower-dimensional cells (see below).

Note that both of the methods just displayed distinguish between $\delta p = 0^+$ and $\delta p = 0^-$. In the method *adj*, the value $\delta p = 0^\pm$ produces those p-cells which share a common $(p \pm 1)$-cell with the *Cell* instance invoking the method. We introduce here the convention that $-0^+ = 0^+$ and $-0^- = 0^-$, which will be convenient later. The value $\delta p = +1$ corresponds to the coface operator, with larger values of δp corresponding to cofaces of cofaces, *etc*. The value $\delta p = -1$ corresponds to the face operator, while algebraically smaller values of δp correspond to faces of faces, and so on.

We also introduce the convention that all 0-cells share a common theoretical (-1)-cell. Thus, in the method *adj* with $p = 0$, the value $\delta p = 0^-$ produces all 0-cells other than the cell instance invoking the method. Similarly, all n-cells share a common theoretical $(n+1)$-cell, where n is the dimension of the complex, and in the method *adj* with $p = n$, the value $\delta p = 0^+$ produces all n-cells other than the cell instance invoking the method. Consequently, for any given value of $p \in \{0, \ldots, n\}$, the method *adj* is well-defined for $\delta p = 0^-$, $\delta p = 0^+$, and for δp equal to any non-zero integer in the interval $[-p, n-p]$.

The class *Cell* has an additional method which permits the addition of $(p-1)$-cells to a p-cell, and as a byproduct, determines the orientation of the p-cell relative to each of the $(p-1)$-cells comprising it:

- *void add(Cell cellpm1, int r_o),* which adds the $(p-1)$-cell identified by *cellpm1* to the *Cell* instance invoking the method, supposed of dimension p. (The method also assigns an orientation to the $(p-1)$-cell, relative to the

Cell instance invoking the method, where the parameter *r_o* may optionally be used to specify the relative orientation of the first added cell.)

Creation of a p-cell involves three steps: declaration of the p-cell using the *Cell* constructor, addition of the p-cell to a complex using the method *add* of the class *Complex*, and assembly of the p-cell by attaching $(p-1)$-cells using the method *add* of the class *Cell* (in which case the $(p-1)$-cells must already have been declared and added to the same complex as the p-cell). It follows from this last parenthetical remark that we must begin by creating the cells of lowest dimension (normally 0-cells), and continue with cells of ever-increasing dimension, until finally the cells of largest dimension have been created.

As indicated in the description of the *add* method of the class *Cell*, it is often useful to be able to specify the *orientation* of cells. In our framework, we work only with *relative* orientations, between p-cells and adjacent $(p+\delta p)$-cells, where $|\delta p| = 1$. The first $(p-1)$-cell added to a p-cell determines the orientation of the p-cell (see [7] for details). Calculation of relative orientation is equivalent to calculating the incidence numbers [10, p. 223] necessary, for example, for the use of Betti numbers. Practical applications are given in [5,7,8], and below.

As mentioned above, the order of the p-cells C_i^p, as a function of i, depends on the order in which the cells were added to the complex. Given this ordering, and coefficients associated with the cells, we may speak of formal sums of p-cells of K with these coefficients [5,7,8,10,14] or equivalently [10, p. 226], vectors whose components are coefficients of p-cells. A *p-chain*, defined over a complex K and a set G, is a formal sum $\sum_i g_i C_i^p$ of p-cells from the complex K, with coefficients $g_i \in G$. A *p-chain* assigns an element of G to each p-cell $C_i^p = K.cell(i,p)$ in the complex K. As in [7], G might be the set of real numbers, a set of vectors, the RGB color space, or any other set, and as already mentioned, we are particularly concerned in this paper with the case where the g_i are themselves *subchains* on K. Interpreting the formal sum as a row vector of dimension $K.nb\_cells(p)$ having components that are coefficients of p-cells, we may write it as $[g_1, \ldots, g_{K.nb\_cells(p)}]$; this is a vector space if G is a field.

The API includes also the class *Chain*, as well as certain convenient derived subclasses of this class, containing methods [5,7,8]. For example, there are methods to return the corresponding complex K, or return the dimension p of the corresponding cells, or permit access to the coefficient g_i, for the chain invoking the method. Thus, in this last case, we may write either

chain2.coefficient(cell2) = Real3(1,2,2) or *chain2[cell2] = Real3(1,2,2)*

to assign the three-dimensional vector $[1,2,2]$ to the g_i associated with a par-

ticular cell *cell2*.

Any method defined on the elements of G is applicable to chains by applying the method to the coefficients of individual cells: we define $(\sum_i g_i C_i^p).m = \sum_i g_i.m\, C_i^p$. Similarly, any binary operation $\otimes$ defined on $G \times G$ can be extended to chains over G. Thus, if $\otimes$ is defined on elements of the class *Type*, we may specify cell-by-cell operations by writing *chainp_a* $\otimes$ *chainp_b*. These principles are generalized to chains of subchains in Subsection 3.2.

An important method introduced in [7] is *dim_inc*, which is associated with a p-chain on an object:

- *Chain_Type dim_inc(int delta_p, Operation op, Boolean is_signed),* which returns a chain (of class *Chain_Type*) of dimension $p + \delta p$, where δp is an integer specified by *delta_p* (the "dimension increment"). The coefficients of the chain are determined by the operation [2] specified by *op*, and relative orientation is taken into account if *is_signed* is *true*.

Here, *Operation* is an enumerated type, by means of which may be specified addition (*plus*), average (*average*), multiplication (*mult*), geometric mean (*geom_mean*), or some other operation implemented by the programmer. (The default options for *op* and *is_signed* are *plus* and *true*, respectively.) Thus, we might write *chain3 = chain1.dim_inc(+2,average,false)* to produce a 3-chain from a 1-chain (the dimension increment is $3 - 1 = 2$), where the coefficients of the 3-chain are obtained by taking the average of the coefficients of the relevant 1-cells (*i.e.*, those adjacent to a given 3-cell), without taking relative orientation into account [3]. The method *dim_inc* generalizes the concepts of *boundary* and *coboundary* [14].

The value of the coefficient h_i corresponding to a cell $C_i^{p+\delta p}$ in the new chain is computed using any commutative operation Ω, defined with *op*, on the coefficients corresponding to p-cells that are adjacent to the cell $C_i^{p+\delta p}$ [7, Sec. 3.4.2]. To illustrate its use in practice, if *chain2_n* is a 2-chain over R^3 representing the vector normals of each planar face of a geometric complex, we may calculate a 0-chain *chain0_n* over R^3 that contains, for each 0-cell, the average of the normals for the adjacent 2-cells, for use in shading algorithms:

chain0_n = chain2_n.dim_inc(-2, average, false)

(the operation over the vector coefficients is the operation *average*, and we do not take relative orientation into account); the vectors may then be normalized, using *chain0_n = chain0_n.normalize()* . The method *dim_inc* also found

[2] The operation *op* must be valid for the coefficients of the method-invoking chain.
[3] This is in fact the only choice when $|\delta p| \neq 1$.

significant use in both of the application examples described in [7], and it will be generalized further in Section 3.

3 Chains of subchains

In this section we describe an extension to the API, for the class G of coefficients: a coefficient g_i of a cell is now permitted to be a *subchain* of a chain. This subchain consists of a vector of coefficients on certain $(p+\delta p)$-cells of the complex, namely, those that are adjacent to C_i^p. We begin with an example.

Consider the mass-spring network illustrated in Figure 1 where, associated with each 2-cell, are four angles defining the rest position of the network. (Equivalently, we may associate one to four angles with each 0-cell. Such equivalences are the basis of a duality to be introduced below.) To model such situations, it is convenient to permit the coefficient g_i of a cell C_i^p to be a *subchain* defined on the $(p+\delta p)$-cells $C_i^p.adj(j, \delta p), j = 1, \ldots, nb\_adj(\delta p)$. Thus, in Figure 1, we might decide to work with $p = 2$ and $\delta p = -2$, corresponding to a 2-chain with coefficients that are subchains defined on the 0-cells $C_i^2.adj(j, -2), j = 1, \ldots, 4$. Let us denote the j'th component of g_i by $g_{i,j}$, *i.e.*, the angle at rest associated with the pair of cells C_i^2 and $C_i^2.adj(j, -2)$ is stored in $g_{i,j}$. For example, in Figure 1, the coefficient $g_{i,3}$ is associated with the pair formed of the 2-cell C_i^2 and the 0-cell $C_i^2.adj(3, -2)$. Similarly, the coefficient $g_{i',1}$ is associated with the pair formed of the 2-cell $C_{i'}^2$ and the 0-cell $C_{i'}^2.adj(1, -2)$. In the example, the 0-cell $C_i^2.adj(3, -2)$ is equal to $C_{i'}^2.adj(1, -2)$.

In general, we define a p-chain of $(p + \delta p)$-subchains on a complex K by a vector of vectors of coefficients $g_{i,j}$:

$$\left[(g_{1,1}, g_{1,2}, \ldots, g_{1,\nu_1^{\delta p}}), (g_{2,1}, g_{2,2}, \ldots, g_{2,\nu_2^{\delta p}}), \ldots, (g_{N_p,1}, g_{N_p,2}, \ldots, g_{N_p,\nu_{N_p}^{\delta p}})\right],$$

where the coefficient $g_{i,j}$ is associated with the pair of cells $[C_i^p, C_i^p.adj(j, \delta p)]$, and where $N_p = K.nb\_cells(p)$, $\nu_i^{\delta p} = C_i^p.nb\_adj(\delta p)$, and $i = 1, \ldots, N_p$. This p-chain of $(p + \delta p)$-subchains is denoted as a (p, q)-chain with $q = p + \delta p$.

It is possible that δp is 0^+ or 0^-. Indeed, we permit $q = p + 0^+$, which gives a $(p, p + 0^+)$-chain, with $\delta p = q - p = p + 0^+ - p = 0^+$, and $p = q + 0^+$, which gives a $(q + 0^+, q)$-chain, with $\delta p = q - p = q - (q + 0^+) = 0^+$ since, as mentioned in Section 2, we have $-0^+ = 0^+$. Both $(p, p + 0^+)$-chains and $(q + 0^+, q)$-chains are made up of chains and subchains involving cells of the same dimension, with the adjacency relationship that specifies the subchains,

associated with a cell, defined by $\delta p = 0^+$. Exactly analogous remarks apply for $(p, p + 0^-)$- and $(q + 0^-, q)$-chains.

A subchain coefficient of C_i^p can be written $(g_{i,1}, g_{i,2}, \ldots, g_{i,\nu_i^{\delta p}})$, and this can be viewed as a formal sum, as in Section 2:

$$\sum_{j=1}^{C_i^p.nb\_adj(\delta p)} g_{i,j}\, C_i^p.adj(j, \delta p).$$

Continuing, the (p, q)-chain can be associated with the formal sum

$$\sum_i \sum_{j=1}^{C_i^p.nb\_adj(\delta p)} g_{i,j}\, C_i^p.adj(j, \delta p).$$

The API provides classes of chains of subchains with coefficients in R, R^2 and R^3; the programmer can add (p, q)-chains with coefficients of class $Type$, by means of a constructor of the form

Chain2_Type(int p, int q, Complex complex_K).

The name *"Chain2"* is a shorthand for "chains of subchains".

If the coefficients $g_{i,j}$ are elements of a field, then the vectors of vectors defining a (p, q)-chain form a vector space of dimension $\sum_{i=1}^{N_p} \nu_i^{\delta_p}$ having a particular sub-space structure. This will be mentioned again in the conclusion.

3.1 Dual chains of subchains

As already mentioned, in the p-chain of $(p + \delta p)$-subchains introduced above, the coefficient $g_{i,j}$ is associated with the pair of cells [4] $[C_i^p, C_i^p.adj(j, \delta p)]$. But adjacency is a symmetric relation [18, p. 156]: if b and c are cells of a complex K such that $b^{p+\delta p} = c^p.adj(i, \delta p)$, then there exists a unique j such that $c^p = b^{p+\delta p}.adj(j, -\delta p)$. (In order to include the possibilities $\delta p = 0^-$ and $\delta p = 0^+$ here, the convention that $-0^+ = 0^+$ and $-0^- = 0^-$ is crucial.) Consequently, associated with every $(p, p+\delta p)$-chain (that is to say, a p-chain with coefficients that are $(p+\delta p)$-subchains), there is a *dual* $(p+\delta p, p)$-chain (that is, a $(p+\delta p)$-chain with coefficients that are p-subchains). There is no restriction on the sign

[4] Note therefore that the order corresponding to the index i is the order of addition of cells to the complex, while the order corresponding to the index j depends on the order of assembly of the cells. Note also that when we wish to refer to a cell without reference to its sequential position, we will use lower-case letters in place of C.

of δp: it is only necessary that $0 \leq p, p+\delta p \leq n$, where n is the dimension of the complex. It is permitted that $\delta p = 0^+$ or 0^-, in which case both the chain and its dual are made up of chains and subchains involving p-cells, with the adjacency relationship that specifies which subchains are associated with a cell defined in both cases by δp: $(p+\delta p) - p = \delta p$ and $p - (p+\delta p) = \delta p$, since $-\delta p = \delta p$.

As an example of dual chains, suppose we have a $(2,0)$-chain *chain20*, similar to the example illustrated in Figure 1, *i.e.*, a 2-chain with coefficients that are 0-subchains. This $(2,0)$-chain might be defined by the vector of vectors

$$[(7,5,2,1),(4,2,3,11),(1,9,-2,2),(3,4,1,7),(1,6,2,-3),(1,10,4,3)]$$

as illustrated in Figure 2. Dual to this $(2,0)$-chain is a $(0,2)$-chain *chain02*,

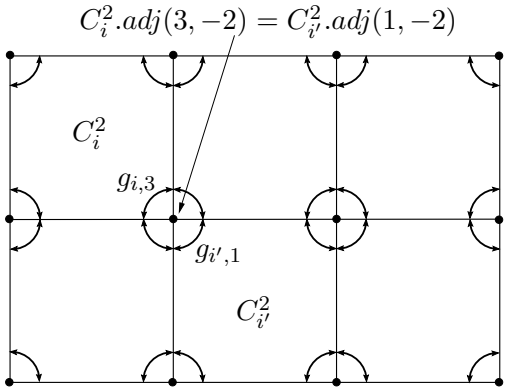

Fig. 1. A 2-complex with rest-position angles.

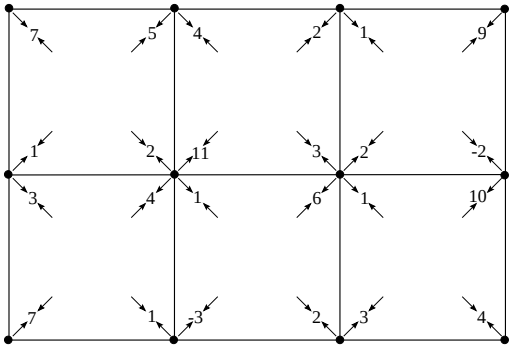

Fig. 2. Illustrating both a $(2,0)$- and a $(0,2)$-chain (dual chains).

a 0-chain with coefficients that are 2-subchains. This $(0,2)$-chain might be defined by the vector of vectors

$$[(7),(5,4),(2,1),(9),(1,3),(2,11,4,1),\ldots]$$

as is also illustrated in Figure 2: the pictures illustrating the two chains are identical. The dual of a $(p, p+\delta p)$-chain is unique and well-defined, due to the symmetry of the adjacency relation, mentioned above.

Our API [8] provides the method *dual* to produce the dual of a chain. Thus, for example, if *chain*20 is a $(2,0)$-chain, then *chain*20.*dual*() is a $(0,2)$-chain which we might denote by *chain*02. Similarly, if *chain*02 is any $(0,2)$-chain, then *chain*02.*dual*() is a $(2,0)$-chain, and *chainpq*.*dual*().*dual*() produces a chain equivalent to *chainpq*.

3.2 Extension of methods and binary operations

We extend the method (of the class *Chain*)

Type& coefficient(int j)

and the equivalent operator [*int j*] [7, Sec. 3.4.1] to subchains as follows: *coefficient(j)* applied to the subchain returns the coefficient of the j'th cell in the subchain, *i.e.*, the j'th coefficient of the corresponding vector. Similarly, the API extends the method *coefficient* to the class *Chain2_Type:* the method

Type& coefficient(Cell cp, Cell cq)

returns the reference to the coefficient $g_{i,j}$ (of class *Type*) associated with the pair of cells *cp* and *cq* of the (p, q)-chain invoking the method. Here, *cp* and *cq*, of dimension p and q, respectively, must be adjacent (in the sense $adj(\delta p)$).

Other methods, and binary operations, are extended as follows. Normally the result of applying a method or binary operation to a p-chain is to apply the method or operation on a cell-by-cell basis, and we extend this concept to subchains: a method or binary operation defined on the coefficients of a q-subchain is defined on the q-subchain by applying the method or operator on a cell-by-cell basis. Then, the extension to a p-chain of $(p + \delta p)$-subchains (*i.e.*, to a (p, q)-chain) can also be made, using the principle introduced in Section 2. A simple example is the operator "+", which is defined on pairs of reals, and therefore on pairs of subchains of reals, and therefore on pairs of (p, q)-chains of reals. The result of the addition of two (p, q)-chains of reals will be a (p, q)-chain of reals. For each pair $[C_i^p, C_i^p.adj(j, \delta p)]$, the sum of the corresponding coefficients is stored in the resulting (p, q)-chain. For example, we might add together two $(2, 0)$-chains *chain20_a* and *chain20_b* by means of *chain20_c = chain20_a + chain20_b* where *chain20_c* is a third $(2, 0)$-chain.

We can also extend methods defined on a collection of coefficients. The methods and binary operators applied to p-chains, described so far, were defined on a cell-by-cell basis to produce another p-chain (see Section 2). Also available, however, are methods defined on a collection of coefficients and which return a single value. For example, the method *sum* returns the sum of all the coefficients of a chain over R^n, $n \geq 1$. This method might be used, for example, to find the geometric mean of the positions associated with the 0-cells of a complex K of the class *Complex_p* [7, Sec. 3.4.3]. To do this, it is sufficient to write *K.chain_p.sum()/K.nb_cells(0)* . (Also, although the API does not deal explicitly with cochains [1, p. 429], they could be introduced by storing their coefficients in a 1-chain. Thus, for example, given a 1-chain *chain1* defined on

the complex K, and another 1-chain *chain1_cochain* containing the coefficients, viewed as a column-vector, of a 1-cochain on the space of 1-chains [12, p. 53], we can compute $\int_{chain1} chain1\_cochain$ by *(chain1_cochain*chain1).sum().* The method *mult* is similar to the method *sum*: it returns the product of the coefficients of a chain over R^n, $n \geq 1$ (in our example we used component-by-component multiplication). These methods have also been extended, in the API, from chains to subchains.

In the context of a (p, q)-chain, methods such as *sum* and *mult* are applied to each of the q-subchains, following the convention introduced in Section 2. Thus, the coefficients of each q-subchain are added or multiplied together, to produce the coefficient g_i of a p-chain. An example is shown in Figure 3, which shows how the method *sum* applied to a certain $(2, 0)$-chain produces a 2-chain. The coefficients of the $(2, 0)$-chain are shown in Figure 3 in the same way they were shown in Figure 2; the curved arrows in Figure 3 indicate how the coefficients of the 2-chain are obtained from those of the $(2, 0)$-chain.

It will be observed that the operation just illustrated, on a chain of subchains, is very similar to the method *dim_inc*. In fact, we will see below that the method *sum* on a (p, q)-chain can be used to obtain the same result as the method *dim_inc* with the operation *plus*.

Another sequence of operations, which is often useful, produces a q-chain *chainq* from a (p, q)-chain *chainpq*:

chainq = chainpq.dual().sum().

An example is given in Figure 4. The coefficients of the $(2, 0)$-chain shown

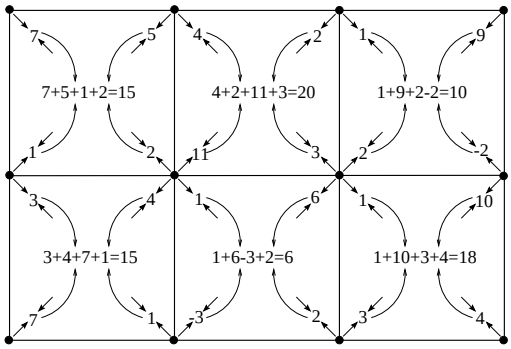

Fig. 3. The *sum* method produces a 2-chain from a $(2, 0)$-chain.

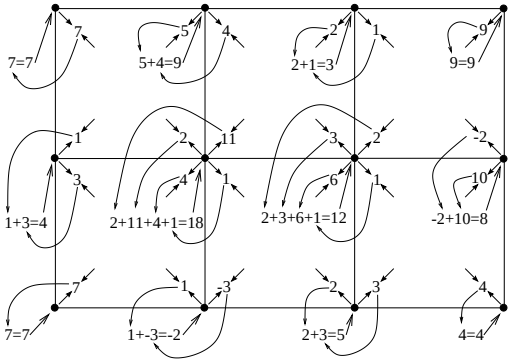

Fig. 4. Methods *dual* and *sum* produce a 0-chain from a $(2, 0)$-chain.

there are the same as those shown in Figure 3: the graphical representation is indistinguishable from that of the (0,2)-chain subsequently produced by the *dual* method. The curved arrows in Figure 4 show how the coefficients of a 0-chain are obtained from those of the $(0, 2)$-chain.

Although the examples here were presented exclusively in terms of the method *sum*, we could as easily have chosen *mult*, which calculates the product of the coefficients (assuming that a product operation is defined on the coefficients): one may use any commutative operation Ψ producing a single-valued result.

3.3 Orientation in terms of subchains

It may sometimes be useful to store the calculated relative orientations between the p-cells of a complex and their faces (see Section 2 and [7, Sec. 3.2]). A convenient way to do so is to store them in a $(p, p-1)$-chain over the integers or the reals: it is sufficient to set each coefficient $g_{i,j}$ to the value of $C_i^p.orientation(C_i^p.adj(j,-1))$. Included in the API is a method, in the class *Complex*, which produces these relative orientations:

- *Chain2_Real orientation(int p).* Returns a $(p, p-1)$-chain of relative orientations.

It will be seen below that this $(p, p-1)$-chain of relative orientations may be useful, in particular, in the manipulation of chains of subchains.

3.4 The methods out *and* in

We now introduce the method *out*, which produces a (p, q)-chain *chainpq* from a p-chain. Given the p-chain and δp as input, the coefficients g_i of the p-chain are simply assigned to $g_{i,j}$ for $j = 1, \ldots, C_i^p.nb\_adj(\delta p)$, and q is taken to be $p + \delta p$. The values $\delta p = 0^+$ and $\delta p = 0^-$ are permitted.

The API includes the method *out*, which is valid for *Chain_Type* classes:

- *Chain2_Type out(int delta_p, Boolean is_signed).* Returns a $(p, p+\delta p)$-chain (of class *Chain2_Type*), given the p-chain that invokes the method. Relative orientation is considered when *is_signed* is *true* (the default option), and δp is specified by *delta_p*.

When relative orientation is taken into account (a valid request only when δp is equal to $+1$ or -1), the $(p, p+\delta p)$-chain is multiplied by the $(p, p-1)$-chain of relative orientations *K.orientation(p)* (when $\delta p = -1$), or by the $(p+1, p)$-chain of relative orientations *K.orientation(p+1)*, to which the method *dual* has first been applied (when $\delta p = +1$). An example illustrating the application of the method *out(-2, false)* on a 2-chain is shown in Figure 5.

The method *in*, also valid for *Chain_Type* classes, returns the $(p + \delta p, p)$-chain

that is the dual of the chain returned by *out(delta_p, is_signed)*:

chainqp = chainp.out(delta_p, is_signed).dual()

with *is_signed* equal to *true* or *false*:

- *Chain2_Type in(int delta_p, Boolean is_signed).* Returns a $(p, p+\delta p)$-chain (of class *Chain2_Type*), the dual of the chain which invoked the method. Relative orientation is taken into account when *is_signed* is *true* (the default option). The value of δp is specified by *delta_p*.

Since the illustrations of a chain and its dual are identical, application of the method *in(-2, false)* to a 2-chain produces the same illustration (Figure 5) as the application of *out(-2, false)* to the same chain. The result, however, is a $(0, 2)$-chain for the method *in*, and a $(2, 0)$-chain for the method *out*.

An example of the application of the method *in(+1, true)* to a 1-chain is shown in Figure 6, and the same illustration serves for the the method *out(+1, true)*: the result is a $(2, 1)$-chain, or a $(1, 2)$-chain, respectively. In this example, we consider relative orientation. For clarity, absolute orientations are shown, but in our framework, it is relative orientations that are actually used. It will be observed in the example that certain of the coefficients of the $(2, 1)$-chain have changed, relative to the sign of the coefficient of the 1-chain, because the relative orientation of the associated pair (a 2-cell and a 1-cell) was -1.

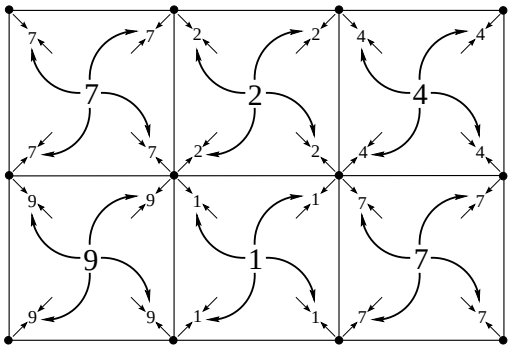

Fig. 5. A $(2, 0)$-chain from a 2-chain.

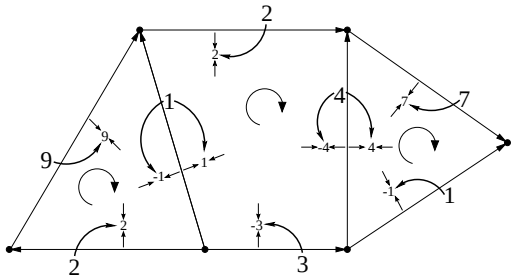

Fig. 6. A $(2, 1)$-chain from a 1-chain using relative orientation.

The methods *out* and *in* are especially useful when we wish to apply binary operations, or other methods, to chains and chains of subchains. For example, we might want to add the coefficient of each 2-cell of a $(2, 1)$-chain *chain21*, and the corresponding coefficient of a 2-chain *chain2*, to produce a $(2, 1)$-chain. This can be done as follows: *chain21 + chain2.out(-1,false).* It will be seen below, in the examples, that this sort of operation is frequently required.

We will list here some other examples, to illustrate valid operations involving chains and chains of subchains. Let *chain0* be a 0-chain, *chain2* a 2-chain,

chain20 a $(2,0)$-chain, and *chain02* a $(0,2)$-chain. We might have:

- *chain20 = chain2.out(-2,false)*;
- *chain20 = (chain2.in(-2,false)).dual()*; (equivalent to the line above)
- *chain20 = chain20 + chain2.out(-2,false)*;
- *chain20 = chain0.in(+2,false)*;
- *chain02 = chain2.in(-2,false)*.

The following equivalences relating the methods *dim_inc* and *in* are valid:

$$chainp.dim\_inc(\delta p,plus,false) \equiv chainp.in(\delta p,false).sum();$$

$$chainp.dim\_inc(\delta p,plus,true) \equiv chainp.in(\delta p,true).sum();$$

the second is valid only for δp equal to $+1$ or -1. Similar equivalences can be established for other operations Ω, such as *mult*, *average*, and *geom_mean*. Thus, the operations *in* and *sum*, which are essential for the manipulation of chains of subchains, are sufficient to replace the method *dim_inc*. In spite of this and other advantages [8, Sec. 4.2] of using *in* combined with an arbitrary commutative operation Ψ, we have nonetheless retained the method *dim_inc*. First of all, it is easy to use, and it is concise. Secondly, *dim_inc* executes much faster, since it is not necessary to allocate memory for the chain of subchains.

3.5 The dim_inc *method applied to chains of subchains*

As just observed, it is possible to use (p,q)-chains to define the method *dim_inc*. On the other hand, we have not yet examined the possibility of applying *dim_inc* itself to a (p,q)-chain. It turns out that this idea is very useful, just as the use of the ordinary version of *dim_inc* was useful in the context of p-chains. In particular, the extended version of *dim_inc* will be applied in Subsection 4.1 in an example involving torques in a spring network.

We generalize the application of *dim_inc* using the principle of Section 2, by which methods can be extended to chains by applying them to the coefficients of the chain, one at a time. Thus, the *dim_inc* method is applied to each of the q-subchains of the (p,q)-chain.

Just as the application of *dim_inc(δp)* to a p-chain with primitive coefficients produces a $(p+\delta p)$-chain, so the application of *dim_inc(δq)* to a q-subchain produces a $(q+\delta q)$-subchain. Thus, application of *dim_inc(δq)* to a (p,q)-chain

chainpq will result in a $(p, q+\delta q)$-chain *chainpq*$_2$, calculated [5] as follows:

$\delta p \leftarrow q - p$;
$\delta p_2 \leftarrow q + \delta q - p$; // $(= \delta p + \delta q)$;
for all cells C_i^p of complex K **do**
  for j=1 **to** $C_i^p.nb\_adj(\delta p_2)$ **do**
    $c^{q+\delta q} \leftarrow C_i^p.adj(j, \delta p_2)$;
    $S \leftarrow C_i^p.adj(\delta p) \sqcap c^{q+\delta q}.adj(-\delta q)$;
    $chainpq_2.coefficient(C_i^p).coefficient(j) \leftarrow$
      $\Omega_{k=1}^{S.length} chainpq.coefficient(C_i^p, S[k])$;

The outer loop treats each cell C_i^p of the complex, one at a time: calculation of the $(q+\delta q)$-subchain associated with this cell is inside this loop. The cells in the vector S are of dimension q, and that the algorithm uses the intersection operation $\sqcap$ between two vectors of cells: $v_1 \sqcap v_2$ contains exactly those cells which are in both v_1 and v_2.

When relative orientation is taken into account (δq must be $+1$ or -1), the righthand side of the last line of the algorithm is replaced by

$$\Omega_{k=1}^{S.length} \sigma \ chainpq\_a.coefficient\ (C_i^p, S[k]),$$

where σ is the relative orientation between $S[k]$ and $c^{q+\delta q}$.

We illustrate the algorithm with the example shown in Figure 7. The input is a $(2,1)$-chain *chain21*, which invokes the method *dim_inc(-1, plus, false)*. The input parameters imply that the operation Ω on the coefficients is addition, and relative orientation is not to be taken into account. The result is a $(2,0)$-chain *chain20*:

chain20 = chain21.dim_inc(-1, plus, false).

For this operation to be realized, *dim_inc* must be applied to each of the 1-subchains, producing 0-subchains. Each of the 1-subchains is a coefficient of a 2-cell (since the input is a $(2,1)$-chain), and each of the resulting 0-subchains is also a coefficient of a 2-cell (since the result is a $(2,0)$-chain). The sum of coefficients for each 0-subchain will involve only 1-subchain-coefficients corresponding to a single 2-cell.

[5] We impose the following conditions on the input (p,q)-chain ($q = p + \delta p$) and the input δq: $0 \leq q + \delta q \leq n$, δp and δq are not equal to 0^+ or 0^-, and $q + \delta q - p$ is not zero.

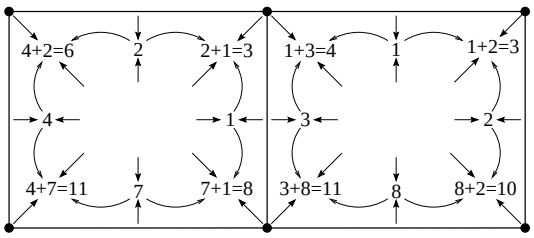

Fig. 7. A $(2,0)$-chain produced from a $(2,1)$-chain by *dim_inc(−1, plus, false).*

4 Example systems with the extended framework

4.1 Mass-spring networks with torques

In our first example, torques will be added to a modified version of the mass-spring network [15, Sec. 2] described in [7, Sec. 4.1.1] (the model here is based on a 2-complex, rather than a 1-complex). First of all, 2-cells will be defined on the complex; then, rest angles will be specified at each (2-cell, 0-cell)-pair for which the 2-cell and the 0-cell are adjacent, and stored in a $(2,0)$-chain.

A user, having constructed a geometric object (a cellular complex) in three dimensions, may wish to embed it in a physical environment, with forces (such as gravity) explicitly modeled, and in such a way that the object will react appropriately after collision with another object. Such an embedding was accomplished with the model of [7, Sec. 4.1.1], for objects such as a tetrahedron and a plane (the "floor"), by associating springs with 1-cells. If, however, it is desired to construct an object such as a cube, then with this model it will be necessary to introduce new 1-cells, corresponding to supplementary springs, so that the object will retain approximately the shape of a cube. An alternative approach is to add torques to the model. If we do this, it is not necessary to add supplementary springs, since if rest angles of $\pi/2$ are assigned, then the cube will retain its approximate form. We have obtained quite satisfactory results with this approach, for several object forms (see [6]).

We begin with a summary of the concept of *torque*. In the remainder of the paper we use boldface characters to denote vectors in R^2 or R^3.

4.1.1 Torque

In Figure 8 there are three masses linked by two springs. A *torque* acting on the mass m_2 is produced at the pivot point when the system is not in its rest position (indicated by the dashed lines in the figure). We have $\boldsymbol{T} = \boldsymbol{r} \times \boldsymbol{F}$, where $\times$ denotes the vector product in a righthand coordinate system, $\boldsymbol{r}$ is the difference between the positions of mass m_1 and mass m_2, and $\boldsymbol{F}$ is the

force applied to the mass m_1. The vector $\boldsymbol{T}$ in Figure 8 points out of the page. In our simulation the torque $\boldsymbol{T}$ present when the system is not at rest will be taken as a starting point, and $\boldsymbol{F}$ will be calculated from (1), below.

We assume that the torque $\boldsymbol{T}$ is orthogonal to the plane P defined by the positions of the three masses (and we assume that they are not collinear). Let $\boldsymbol{v}_n = \frac{\boldsymbol{r}\times\boldsymbol{r}'}{|\boldsymbol{r}\times\boldsymbol{r}'|}$, where $\boldsymbol{r}'$ is the difference between the positions of mass m_2 and mass m_3. Thus, $\boldsymbol{v}_n$ is a unit vector orthogonal to the plane P. We make the hypothesis that the torque is given by $\boldsymbol{T} = \boldsymbol{v}_n \cdot [k_a(\cos\theta - \cos\theta_r)]$ where θ is the angle between the two springs, θ_r is the angle between the two springs when the system is at rest, and k_a is a torque constant (for a given value of $\cos\theta - \cos\theta_r$, the larger the value of k_a the larger the magnitude of the torque). We have therefore $|\boldsymbol{T}| = |k_a(\cos\theta - \cos\theta_r)|$.

Normally there will be many vectors $\boldsymbol{F}$, lying in the plane P, such that $\boldsymbol{T} = \boldsymbol{r} \times \boldsymbol{F}$; we will take the solution orthogonal to $\boldsymbol{r}$. Given this, we have [8, p. 31]

$$\boldsymbol{F} = \frac{\boldsymbol{T} \times \frac{\boldsymbol{r}}{|\boldsymbol{r}|}}{|\boldsymbol{r}|} \ . \tag{1}$$

Thus, given $\boldsymbol{T}$ and $\boldsymbol{r}$, we can find the force $\boldsymbol{F}$ on the mass m_1. We can also apply the same formula at the position of mass m_3, using $\boldsymbol{r}'$ and $\boldsymbol{F}'$ in place of $\boldsymbol{r}$ and $\boldsymbol{F}$.

The sum of the forces in the system must always be zero, and the forces applied to the mass m_2 are $-\boldsymbol{F}$ and $-\boldsymbol{F}'$. Thus, as can be observed from Figure 8, there are four forces $(\boldsymbol{F}, -\boldsymbol{F}, \boldsymbol{F}', -\boldsymbol{F}')$ that sum to zero. In a simulation of this system, we want to propagate these forces to each (2-cell, 0-cell)-pair for which the 2-cell is adjacent to the 0-cell. (Here, the mass m_2 in Figure 8 should be visualized as lying at the 0-cell, and the line segments corresponding to $\boldsymbol{r}$ and $\boldsymbol{r}'$ as forming a corner of the 2-cell.) The associated torques are stored in a $(2, 0)$-chain, and the methods introduced above, for chains of subchains, permit description of the propagation of the forces resulting from the torques.

4.1.2 Torques using chains

We first construct the 2-complex, ensuring that whenever two 2-cells C_i^2 and C_j^2 are incident to a common 1-cell C_k^1, they have opposite relative orientation with respect to the 1-cell: $C_i^2.\mathit{orientation}(C_k^1) = -C_j^2.\mathit{orientation}(C_k^1)$. This is necessary below, to guarantee that the force vectors $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ are correctly oriented.

Our goal now is to compute the 0-chain of the sums of forces $\boldsymbol{F}$ that result from the torques in the network, so that we can later add these sums of forces

$\boldsymbol{F}$ to the other forces acting on a 0-cell of the network. In summary, we will

- compute the vectors $\boldsymbol{r}$, directed according to the orientation of the 1-cells;
- compute the cosines of the angles θ;
- compute the differences between the cosine of the current angle and cosine of the rest angle: $\cos\theta - \cos\theta_r$;
- compute the torques $\boldsymbol{T}$ and store them in a $(2,0)$-chain;
- compute certain sums of forces resulting from the torques, and store them in a $(2,1)$-chain;
- compute certain sums of forces resulting from the torques, and store them in a $(2,0)$-chain;
- compute the sums of forces $\boldsymbol{F}$ resulting from the torques and acting on a single 0-cell, and store them in a 0-chain.

We first use the API to calculate the 1-chain *chain1_r* (over R^3) containing the vector $\boldsymbol{r}$ for each 1-cell, using the position and relative orientation of its 0-cells:

chain1_r = network.chain0_p.dim_inc(+1, plus, true)

where *network.chain0_p* holds the geometric positions of the 0-cells.

We also calculate the normalized vectors, and store them in a $(2,1)$-chain over R^3:

chain21_r_norm = (chain1_r.normalize()).in(+1, true) ,

which has the effect of modifying the vector associated with a 1-cell, depending on the relative orientation of this 1-cell with the adjacent 2-cell. (Similarly, we calculate *chain21_r = chain1_r.in(+1, true)* .) Consider for example the vector pointing *vertically downwards* from the upper lefthand 0-cell in Figure 9. The corresponding coefficient (another vector) of the $(2,1)$-chain is shown just

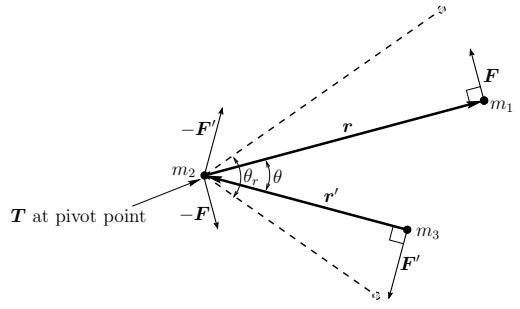

Fig. 8. Torque forces and rest-position angles.

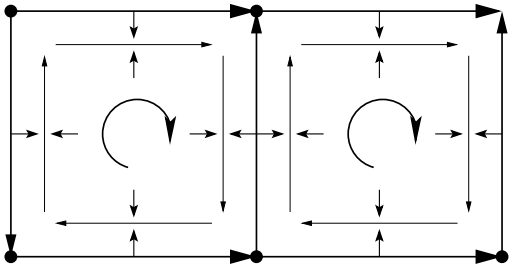

Fig. 9. A $(2,1)$-chain computed from *chain1_r_norm*.

to the right of the original vector, and it points upwards, since the relative

orientation between the 1-cell and the 2-cell is -1. On the other hand, if we consider the vector pointing *horizontally to the right* from the upper lefthand 0-cell, then the corresponding coefficient of the $(2,1)$-chain (shown just below the horizontal vector) is another horizontal vector pointing to the right, since the relative orientation between the 1-cell and the 2-cell is $+1$.

Given the $(2,1)$-chain *chain21_r_norm*, we can calculate the cosines of the angles θ with the statement

chain20_cos_theta = −chain21_r_norm.dim_inc(-1, mult, false).sum_components()

which produces a $(2,0)$-chain from the input $(2,1)$-chain. The operation Ω specified here is component-by-component multiplication, without consideration of relative orientation, and it is followed by application of the method *sum_components*, which simply sums the components of a vector. These operations (together with the minus sign in the displayed statement, above) calculate the inner product between $\boldsymbol{r}$ and $-\boldsymbol{r}'$ (see Figure 8), and thus, since the vectors are normalized, the cosine of the angle between them.

We next calculate $\cos\theta - \cos\theta_r$, and store the result:

chain20_diff_cos_theta = chain20_cos_theta - chain20_cos_theta_rest

where the last-mentioned chain (giving the value of $\cos\theta_r$ for each adjacent (2-cell, 0-cell)-pair) has been precalculated. We then calculate the $(2,0)$-chain of torques:

*chain20_T = chain20_vn*chain20_ka*chain20_diff_cos_theta* .

The computation of *chain20_vn*, which for each adjacent (2-cell, 0-cell)-pair contains the vectors $\boldsymbol{v}_n$ orthogonal to the plane P of Figure 8, will be discussed below. The chain *chain20_ka* contains the corresponding torque constant k_a for each adjacent (2-cell, 0-cell)-pair.

The next step is to propagate the four forces that result from the individual torques. But, before doing so, let us examine in detail the necessary calculations in a 2-cell, for each 1-cell. There are two 0-cells c_1^0 and c_2^0 adjacent to the 1-cell, and torques $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, respectively, are calculated at c_1^0 and c_2^0, which give rise to the forces $\boldsymbol{F}_1$ and $\boldsymbol{F}_2$. The forces $-\boldsymbol{F}_1$ and $\boldsymbol{F}_2$ must be applied at c_1^0, and the forces $-\boldsymbol{F}_2$ and $\boldsymbol{F}_1$ must be applied at c_2^0. See Figure 8.

Suppose for example that the relative orientation between the 1-cell and c_1^0 is $+1$, and that between the 1-cell and c_2^0 it is -1. In this case we would proceed

as follows:

(1) add the torques $\boldsymbol{T}_1$ and $\boldsymbol{T}_2$, taking relative orientation into account, producing $\boldsymbol{T}_1 - \boldsymbol{T}_2$ (which we associate with the 1-cell);
(2) change the sign of the sum of torques, producing $\boldsymbol{T}_2 - \boldsymbol{T}_1$;
(3) calculate the sum of forces $\boldsymbol{F}_2 - \boldsymbol{F}_1$ from $\boldsymbol{T}_2 - \boldsymbol{T}_1$;
(4) apply this sum to the 0-cells, taking relative orientation into account.

We thus obtain the desired result, namely, $\boldsymbol{F}_2 - \boldsymbol{F}_1$ for the 0-cell c_1^0 and $\boldsymbol{F}_1 - \boldsymbol{F}_2$ for the 0-cell c_2^0. (Note, however, that a 0-cell has two adjacent 1-cells, and the sum of forces is therefore applied from *both* adjacent 1-cells.)

In order to apply all of these various forces to each 0-cell, we use two $(2,0)$-chains and one $(2,1)$-chain. First of all, we add the torques from the $(2,0)$-chain *chain20_T* into the $(2,1)$-chain

chain21_T = −chain20_T.dim_inc(+1, plus, true)

which produces the sum of the torques $\boldsymbol{T}$ for each adjacent (2-cell, 0-cell)-pair, corresponding to the first two of the four steps in the list above. We then calculate the sum of forces for each 1-cell (step 3 in the list):

chain21_F = (chain21_T ∧ chain21_r_norm)/chain21_r.norm()

where, in our framework, the *norm* method calculates the ordinary Euclidean norm of a vector, and the '∧' operator invokes vector product. The displayed expression corresponds exactly to (1). The $(2,1)$-chain *chain21_F* can now be transferred to the $(2,0)$-chain *chain20_F*, the sum of sums of forces $\boldsymbol{F}$ resulting from the torques for each adjacent (2-cell, 0-cell)-pair. In the API, it suffices to write

chain20_F = chain21_F.dim_inc(-1, plus, true) ;

this operation corresponds to step 4 in the list above.

We now have four forces (the sum of two sums of forces) for each (0-cell, 2-cell)- pair, resulting from the torques associated with this (0-cell, 2-cell)-pair. They can now be combined into a 0-chain by writing

chain0_F = chain02_F.dual().sum()

so that *chain0_F* holds the forces associated with each 0-cell. To complete the process, we add these forces to the other forces on the 0-cells [7, Sec. 4.1]:

chain0_f = chain0_f_spring + chain0_f_g + chain0_f_damp + chain0_F .

It remains only to discuss the calculation of the vectors $\boldsymbol{v}_n$. We were unable to find an expression using chains and chains of subchains to produce the desired $(2,0)$-chain *chain20_vn* containing the vectors $\boldsymbol{v}_n$ associated with each (2-cell, 0-cell)-pair. Instead, this chain of subchains was computed within a double loop [8, p. 39].

Examples of results obtained with this model, for several object forms, are available on the first author's web-site [6].

4.2 Modeling of fluid

We have also applied our approach to the modeling of fluids. This may include gases, such as air. We give only a bare outline here: a complete description can be found in [8, Sec. 4.2].

The fluid model is constructed in two dimensions, but one of the advantages of the chain-model formulation is that it would be relatively easy to extend the simulation to three dimensions. The geometry of the complex is defined using a two-dimensional geometric complex [7,8] and triangular 2-cells. In contrast to [9] which use square 2-cells of constant size, we use triangular 2-cells that permit higher-precision subdivision in certain areas of the domain (adaptive refinement), and that permit also a more accurate approximation of boundaries of obstacles.

Another difference between our method and [9] is that we record the fluid mass associated with each 2-cell (in a 2-chain). This is avoided in [9] by requiring that the quantity of fluid entering a 2-cell must be equal to the quantity leaving, but this requires iterating several times on each 2-cell (between 3 and 6 times, according to [9]). Our approach requires no iteration, but this implies that for certain 2-cells, inflow may temporarily be different from outflow. Thus, in effect, we are permitting a slight compression of the fluid. While in [9] the Navier-Stokes equations for incompressible fluids were used, our method produces results that are visually similar, and this is an appropriate criterion to use in a computer-graphics application.

We begin with the principle that each 2-cell contains a certain fluid mass with a certain average speed, and associate the mass m, and the average velocity $\boldsymbol{v}_a$, as coefficients of various chains at time t. Mass-density is then calculated as m/s, where s is the area of the triangular surface. If temperature is assumed constant, and gravitational force is ignored, then pressure is proportional to mass density. In fact, in our simulations, we have simply taken pressure to be *equal* to the mass-density m/s.

In addition, we use four principles to determine the quantities associated with

the 2-cells at time $t + \Delta t$:

(1) the fluid mass in a 2-cell moves in the direction $\boldsymbol{v}_a$, and part of it is transferred to neighboring [6] cells;
(2) part of the fluid mass in a 2-cell is displaced towards neighboring 2-cells having lower pressure;
(3) there is friction between a 2-cell and its neighboring 2-cells, as well as internal friction within a 2-cell;
(4) the direction of the average speed $\boldsymbol{v}_a$ within a 2-cell tends to follow the boundary of the object [7].

The heuristic is first defined for a single 2-cell and (when relevant) its neighbors, and then applied globally, on the entire complex, by using chains of subchains. This turns out to be very convenient for most parts of the simulation. For example, the calculation of the area of triangles fits nicely with the use of chains of subchains by means of the standard formula

$$s = \sqrt{w(w-a)(w-b)(w-c)}, \quad \text{with} \quad w = \tfrac{a+b+c}{2}, \tag{2}$$

where a, b and c specify the lengths of the sides. We computed the 2-chain *chain2_s* containing the area of the triangles with:

chain1_L = chain1_r.norm()
chain2_w = Chain_Real(0.5) chain1_L.dim_inc(+1, plus, false)*
chain21_L = chain1_L.in(+1, false)
chain21_w = chain2_w.out(-1, false)
chain2_wL = (chain21_w − chain21_L).mult()
*chain2_s = (chain2_w*chain2_wL).squareroot()* .

Details are given in [8, p. 48].

As another example, amongst many, updating the average velocity must be done according to the following formula, which is based on the idea that as part of the fluid mass leaves a 2-cell, due to a difference in pressure, there will be an effect on the velocity of the fluid in the cell:

$$\boldsymbol{v}_a \longleftarrow \frac{m\boldsymbol{v}_a + \sum m^i_{p\_out}\boldsymbol{v}^i_p}{m + \sum m^i_{p\_out}}. \tag{3}$$

Here, $m^i_{p\_out}$ is the amount of fluid mass that is displaced from the 2-cell under study to the neighboring 2-cells c^2_i, via the 1-cells c^1_i (these quanti-

[6] The cells c_1 and c_2 are neighbors if there exists a j such that $c_1 = c_2.adj(j, 0^-)$.
[7] All 1-cells with only one adjacent 2-cell are in the boundary of the object.

ties have been calculated in a previous step, and are accessible in the $(2,1)$-chains *chain12_mpouti.dual()*); also, $\boldsymbol{v}_p^i$ is the additional velocity acquired by this fluid mass as it crosses the 1-cell c_i^1 (accessible in the $(2,1)$-chains *chain12_vpi.dual()*) [8, p. 45]. In our notation, this operation is effected globally over the chain in a very simple way. We first compute the term $\sum_i m_{p\_out}^i$ of equation (3):

chain2_sum2 = chain12_mpouti.dual().sum()

storing the result in a 2-chain over R, and then the term $\sum_i m_{p\_out}^i \boldsymbol{v}_p^i$:

*chain2_sum1 = (chain12_mpouti*chain12_vpi).dual().sum()*

this time storing the result in a 2-chain over R^2. Finally, we compute (3):

*chain2_va = (chain2_m*chain2_va + chain2_sum1)/(chain2_m + chain2_sum2).*

On the other hand, we were unable (with the tools defined in this paper), to express the computations related to the fourth principle, above, in a concise way, without using a loop over all the 1-cells in the complex.

5 Conclusion

In this paper, we have described a framework, based on the concept of chains, for the specification and manipulation of models of systems. The framework permits the formulation of problems and subproblems with ease, and without ambiguity. It also permits a global view of the problem that permits the user to neglect low-level details, to unify the description of a large variety of applications, and to unify the geometric and non-geometric aspects of a system.

The framework described here refines the use of chain models in at least three ways: we focus on the distinction between the geometric and the topological complex, we use a generalized version *(dim_inc)* of the chain operators *boundary* and *coboundary*, and we permit the use of subchains as coefficients. We have also shown how to deal with cell orientation, for cells of arbitrary dimension, using only relative orientation.

An important area for future research is to establish the exact relationship between these refined or extended chain models and classical mathematical constructs. In particular, this would involve study of the subspace structure

of the vector space corresponding to chains of subchains (mentioned in Section 3), the related study of the mathematical significance of the dual operation introduced in Section 3.1, and the study of how our adjacency operations relate to classical topological concepts such as *star* and *link*, and to differential and integral operations [1,4]. It would also be of interest to establish the relationship between these extended chain models and standard physical models, in analogy to [4].

In the context of use of chain models as a descriptive tool, future work in this area should include the study of further methods useful in the context of computer graphics and other areas, which permit the calculation of relevant quantities without the use of explicit loops over sets of cells. Finally, it would be desirable, if possible, to give a characterization of the problems for which the class of methods is appropriate.

References

[1] P. Bamberg and S. Sternberg, A Course in Mathematics for Students of Physics, vols. 1 and 2 (Cambridge University Press, 1990).

[2] F. Bernardini, V. Ferrucci and A. Paoluzzi, Working with dimension-independent polyhedra, Report RAP.07.91, Universitá Degli Studi di Roma "La Sapienza", Dipartimento di Informatica e Sistemistica, 1991.

[3] E. Catmull and J. Clark, Recursively generated B-spline surfaces on arbitrary topological meshes, Computer-Aided Design 10 (1978) 350-355.

[4] J. A. Chard and V. Shapiro, A multivector data structure for differential forms and equations, Mathematics and Computers in Simulation 54 (2000) 33-64.

[5] R. Egli, Cadre de travail pour la spécification de systèmes avec des chaînes sur un complexe cellulaire, PhD thesis, Département d'informatique et de recherche opérationnelle, Université de Montréal, May 2000.

[6] R. Egli, web-site. *http://www.dmi.usherb.ca/~egli/chains.*

[7] R. Egli and N. F. Stewart, A framework for system specification using chains on cell complexes, Computer-Aided Design 32 (2000) 447-459. The article appeared earlier in: Computer-Aided Design 31 (1999) 669-681, but was reprinted in full due to a production error.

[8] R. Egli and N. F. Stewart, Graphical Simulation Using Chain Models, Technical Report #1174, Département IRO, Université de Montréal, July 18 2000: *http://www.iro.umontreal.ca/~stewart.*

[9] N. Foster and D. Metaxas, Realistic animation of liquids, in: Proc. Graphics Interface '96 (Toronto, 1996) 204-212.

[10] J. G. Hocking and G. S Young, Topology (Addison-Wesley, 1961).

[11] P. Lienhardt, Topological models for boundary representation: a comparison with n-dimensional generalized maps, Computer-Aided Design 23 (1991) 59-82.

[12] C. Mattiussi, The finite volume, finite element, and finite difference methods as numerical methods for physical field problems, Advances in Imaging and Electron Physics 113 (2000) 1-146.

[13] R. S. Palmer, Chain models and finite element analysis: an executable Chains formulation of plane stress, Computer Aided Geometric Design 12 (1995) 733-770.

[14] R. S. Palmer and V. Shapiro, Chain models of physical behavior for engineering analysis and design, Research in Engineering Design 5 (1993) 161-184.

[15] X. Provot, Deformation constraints in a mass-spring model to describe rigid cloth behavior, in: Proc. Graphics Interface '95 (Québec, 1995) 147-154.

[16] P. Prusinkiewicz, F. F. Samavati, C. Smith and R. Karwowski, L-system Description of Subdivision Curves, International Journal of Shape Modeling 9 (2003) 41-59.

[17] A. A. G. Requicha, Mathematical models of rigid solid objects, Technical Memo TM-28, University of Rochester, 1977.

[18] J. R. Rossignac and M. A. O'Connor, SGC: a dimension-independent model for pointsets with internal structures and incomplete boundaries, in: M. Wozny et al, eds., Geometric Modeling for Product Engineering (IFIP, North-Holland, Amsterdam, 1990) 145-180.

[19] J. P. Roth, An application of algebraic topology to numerical analysis: On the existence of a solution to the network problem, Proc. Nat. Acad. Sci. 41 (1955) 518-521.

[20] J. P. Roth, Existence and uniqueness of solutions to electrical network problems via homology sequences, in: SIAM-AMS Proceedings, Vol. III, (American Mathematical Society, 1971) 113-118.

[21] E. Tonti, The reason for analogies between physical theories, Applied Mathematical Modeling 1, Polytec di Milano (1976) 37-50.