Scuola Normale Superiore

CLASSE DI SCIENZE

Tesi di Perfezionamento in Matematica

Computational techniques in Graph Homology of the Moduli Space of Curves

Candidato:
Riccardo Murri

Relatori:
Prof. Enrico Arbarello
Dott. Domenico Fiorenza

Anno Accademico 2011/2012

A mio nonno Carlo.

Contents

List of Figures

List of Tables

List of Algorithms

List of Acronyms

BSP	Bulk Synchronous Parallel (see [64])
CPU	Central Processing Unit
FIFO	First In, First Out
GCC	GNU C Compiler
GEPP	Gaussian Elimination with Partial Pivoting
IP	Internet Protocol
MPI	Message Passing Interface (see [48, 49])
OS	Operating System
PU	Processing Unit
SIMC	Sparse Integer Matrices Collection (see [15])
SLES	SuSE Linux Enterprise Server
SMSCG	Swiss Multi-Science Computational Grid (see [58])
SMS	Sparse/Symbolic Matrix Storage (see [15])
TBB	Threading Building Blocks (see [35, 36])
TCP	Transmission Control Protocol
UZH	University of Zurich

CHAPTER 1

Introduction

The object of this thesis is the automated computation of the rational (co)homology of the moduli spaces of smooth marked Riemann surfaces $\mathcal{M}_{g,n}$. This is achieved by using a computer to generate a chain complex, known in advance to have the same homology as $\mathcal{M}_{g,n}$, and explicitly spell out the boundary operators in matrix form. As an application, we compute the Betti numbers of some moduli spaces $\mathcal{M}_{g,n}$.

Our original contribution is twofold. In Chapter 3, we develop algorithms for the enumeration of fatgraphs and their automorphisms, and the computation of the homology of the chain complex formed by fatgraphs of a given genus g and number of boundary components n.

In Chapter 4, we describe a new practical parallel algorithm for performing Gaussian elimination on arbitrary matrices with exact computations: projections indicate that the size of the matrices involved in the Betti number computation can easily exceed the computational power of a single computer, so it is necessary to distribute the work over several processing units. Experimental results prove that our algorithm is in practice faster than freely available exact linear algebra codes.

An effective implementation of the fatgraph algorithms presented here is available at http://code.google.com/p/fatghol. It has so far been used to compute the Betti numbers of $\mathcal{M}_{g,n}$ for $(2g+n) \leqslant 6$.

The Gaussian elimination code is likewise publicly available as open-source software from http://code.google.com/p/rheinfall.

1. Fatgraphs and the homology of the moduli space of Riemann surfaces

In the seminal papers [39] and [40], M. Kontsevich introduced "Graph Homology" complexes that relate the stable homology groups of certain infinite-dimensional Lie algebras to various other topological objects. In particular, the "associative operad" variant of this construction results in a chain complex whose homology is isomorphic to the (co)homology of the moduli space of smooth Riemann surfaces $\mathcal{M}_{g,n}$: the graded module underlying the complex is freely generated by the set $\mathcal{R}_{g,n}$ of fatgraphs of genus g and number of boundary components n, endowed with the differential defined by edge contraction.

A fatgraph[1] is a multigraph enriched with the assignment, at each vertex v, of a cyclic order of the edges incident to v. Such graphs can be "fattened" into a Riemann surface, by gluing polygons along the fatgraph edges in such a way that

[1]Fatgraphs have appeared independently in many different areas of mathematics: several equivalent definitions are known, with names such as "ribbon graphs", "cyclic graphs", "maps", "dessins d'enfants", and "rotation systems". See [42] for a comprehensive survey.

two adjacent edges on the polygon boundary are consecutive in the cyclic order at the common endpoint. The resulting Riemann surface is naturally marked, by choosing the marking points to be the centers of the polygons. There is thus a functorial correspondence between fatgraphs and marked Riemann surfaces; a fatgraph G is said to have genus g and n boundary components if it corresponds to a punctured Riemann surface $S \in \mathcal{M}_{g,n}$.

Chapter 1 presents a construction of Kontsevich' fatgraph complex, deriving it as a relative of Harer's arc-system complex [30, 31], and proves the isomorphism of its homology with the rational (co)homology of $\mathcal{M}_{g,n}$. The main ingredient of this construction is a cell decomposition of $\mathcal{M}_{g,n}$ based on a theorem of Jenkins [37] and Strebel [59]. This fatgraph complex is the same complex that one gets by applying Kontsevich' construction in the associative case; however, the proof given here is specific to $\mathcal{M}_{g,n}$ and does not trivially extend to other cases. The techniques devised by Kontsevich are instead suitable to further generalization to any "modular operad" [18].

A construction of the graph complex, closely following Kontsevich' original work, has been detailed by Conant and Vogtmann in [11]; Hamilton and Lazarev gave a new proof in [27]. Other graph homology complexes related to the (co)homology of $\mathcal{M}_{g,n}$ have also been proposed; most relevant to the subject of this thesis is the work by Godin [23], who constructed a variant complex that computes the *integral* homology of mapping class groups of surfaces with boundary, and gives the homology of $\mathcal{M}_{g,n}$ as a particular case. In the recent preprint [56], LaFountain and Penner constructed a space which is homotopy equivalent to the Deligne-Mumford compactification $\overline{\mathcal{M}}_{g,n}$ and admits a cellularization indexed by suitably decorated fatgraphs.

The fatgraph cellularization of the moduli space of smooth pointed Riemann surfaces and related topics have been extensively studied; the interested reader can find comprehensive accounts of the subject in [1, Chapters XVIII and XIX] and [51].

2. Effective computation of the fatgraph complex

Chapter 3 is concerned with finding an effectively computable representation of fatgraphs, and presenting algorithms to:

(1) compute automorphisms of any given fatgraph (Section 2);
(2) generate the set $\mathcal{R}_{g,n}$ of fatgraphs, given the genus g and number of boundary components n (Section 3);
(3) compute the homology of the fatgraph complex $\mathcal{R}_{g,n}$ (Section 4).

Note that, in contrast with other computational approaches to fatgraphs (e.g., [57]), which draw on the combinatorial definition of a fatgraph, our computer model of fatgraphs is directly inspired by the topological definition, and the algorithm for enumerating elements of $\mathcal{R}_{g,n}$ is likewise backed by a topological procedure.

Theorem 2.24 provides an effective way to compute the (co)homology of $\mathcal{M}_{g,n}$. The Betti numbers of $\mathcal{M}_{g,n}$ can be computed from the knowledge of the dimension of chain spaces W_p of the fatgraph complex and the ranks of boundary operators D_p; this computation is accomplished in the following stages:

I. Generate the basis set of W_*; by definition, the basis set is the set $\mathcal{R}_{g,n}$ of *oriented marked* fatgraphs that correspond to surfaces in $\mathcal{M}_{g,n}$.

II. Work out the differential $D\colon W_* \to W_*$ explicitly as matrices $\boldsymbol{D}^{(p)}$ mapping coordinates in the fatgraph basis of W_p into coordinates relative to the fatgraph basis of W_{p-1}.
III. Compute the ranks of the matrices $\boldsymbol{D}^{(p)}$.

Stage I needs just the pair g, n as input; its output is the set of orientable marked fatgraphs belonging in $\mathcal{R}_{g,n}$. By definition, marked fatgraphs are decorated abstract fatgraphs, whose decoration is a simple combinatorial datum (namely, a bijection of the set of boundary cycles with the set $\{1, \dots, n\}$): therefore, the problem can be reduced to enumerating abstract fatgraphs. With a recursive algorithm, one can construct *trivalent* $\mathcal{R}_{g,n}$-fatgraphs from trivalent graphs in $\mathcal{R}_{g-1,n}$ and $\mathcal{R}_{g-1,n+1}$. All other graphs in $\mathcal{R}_{g,n}$ are obtained by contraction of non-loop edges.

The differential D has a simple geometrical definition: $D(G)$ is a sum of graphs G', each gotten by contracting a non-loop edge of G. A simple implementation of Stage II would just compare each contraction of a graph with p edges with any graph with $p-1$ edges, and score a ± 1 (depending on the orientation) in the corresponding entry of the matrix $\boldsymbol{D}^{(p)}$. However, this algorithm has quadratic complexity, and the large number of graphs involved makes it very inefficient already for $\mathcal{M}_{0,5}$. The simple observation that contraction of edges is defined on the topological fatgraph underlying a marked fatgraph allows us to apply the naive algorithm to topological fatgraphs only, which cuts complexity down by a factor $O((n!)^2)$. The resulting matrix is then extended to marked fatgraphs by the action of graph automorphism groups on the markings of boundary cycles. This is the variant detailed in Section 4.

Stage III is conceptually the simplest: by elementary linear algebra, the Betti numbers can be computed from the rank of matrices $\boldsymbol{D}^{(p)}$ and the dimension of their domain space. The computational problem of determining the rank of a matrix has been extensively studied; it should be noted, however, that this step can actually be the most computationally burdening.

Explicit generators of the homology modules could be computed with a little variant in the last step of the algorithm; however, this is not interesting in connection with the homology of $\mathcal{M}_{g,n}$, since expression of a fatgraph homology class in terms of the "natural" algebro-geometric classes has proved to be a difficult problem [**50**, **33**, **34**], and to-date lacks a general solution.

A implementation of the algorithms presented in this paper has been actually used to compute the Betti numbers of all $\mathcal{M}_{g,n}$ with $2g+n \leqslant 6$. Results are summarized in Table 1.1; see Section 5 for implementation-specific details and a discussion of performance.

All the Betti numbers were already known; the output of the program corroborates computations previously published in the literature. The original sources are scattered across a wide array of publications. For $g \geqslant 1$, the groups $H^1(\mathcal{M}_{g,n}, \mathbb{Q})$ are known from the works of Mumford [**53**] and Harer [**29**]; $H^2(\mathcal{M}_{g,n}, \mathbb{Q})$ has been computed also by Harer in [**29**]; a comprehensive statement with a new proof is given by Arbarello and Cornalba in [**2**] (where a minor mistake in Harer's statement is corrected). The complete *integral* homology of $\mathcal{M}_{1,2}$ and $\mathcal{M}_{2,1}$ has been published in Godin's paper [**23**]. The homology of the $\mathcal{M}_{0,*}$ spaces is computed in [**20**, Corollary, 3.10]; see [**46**] for alternative approaches using results from [**62**] to compute the Poincaré polynomial of $\mathcal{M}_{0,n}$. The Poincaré-Serre polynomial of $\mathcal{M}_{2,2}$ follows as a special case of Corollary III.2.2 in Tommasi's thesis [**61**]; the results also follows by combining [**21**, p. 22] with [**32**, Appendix A]. The rational cohomology

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}
$\mathcal{M}_{0,3}$	1												
$\mathcal{M}_{0,4}$	1	2											
$\mathcal{M}_{0,5}$	1	5	6										
$\mathcal{M}_{0,6}$	1	9	26	24									
$\mathcal{M}_{1,1}$	1												
$\mathcal{M}_{1,2}$	1												
$\mathcal{M}_{1,3}$	1			1									
$\mathcal{M}_{1,4}$	1			4	3								
$\mathcal{M}_{2,1}$	1		1										
$\mathcal{M}_{2,2}$	1		2			1							

TABLE 1.1. Betti numbers of $\mathcal{M}_{g,n}$ for $2g+n \leqslant 6$. For readability, null values have been omitted and the corresponding entry left blank.

of $\mathcal{M}_{1,4}$ is completely described (as a particular case) by Theorem 1 of [25]; it can also be explicitly computed by using the tools developed by Getzler in [22]. In all these cases, the numerical results agree with the values in Table 1.1.

The complete set of Betti numbers has apparently been completely computed for just a few spaces, besides those presented in Table 1.1. The equivariant Serre polynomials of $\mathcal{M}_{g,n}$ with $2g + n \leqslant 7$ are completely tabled in [19]. Only for $g = 0$ and $g = 1$ is the Poincaré polynomial of $\mathcal{M}_{g,n}$ known for any $n > 0$; as pointed out earlier, for $\mathcal{M}_{0,*}$ the result is due to Getzler [20], but it can be derived from other results as shown in [46]; a complete description of the cohomology of $\mathcal{M}_{1,*}$ is contained in the preprint [25] by Gorinov. The Betti numbers of $\mathcal{M}_{2,n}$ are well-known for $n \leqslant 2$; for $n = 1$ see, e.g., Godin's [23], but see also [46] for alternative approaches; for $n = 2$, see [21, 32, 61]; O. Tommasi has announced complete results for $\mathcal{M}_{2,3}$ and $\mathcal{M}_{2,4}$ but they have not been published yet. The rational cohomology of $\mathcal{M}_{3,1}$ is given in [6] (which refines results from [19, 44]); the one of $\mathcal{M}_{3,2}$ has been computed by Tommasi in [60]. No other computation of Betti numbers of the spaces $\mathcal{M}_{g,n}$ is known to the author; an online public attempt to gather information about the known Betti numbers of $\mathcal{M}_{g,n}$ is ongoing at [46].

Along with the computation, the entire family of fatgraphs $\mathcal{R}_{g,n}$ (with $2g + n \leqslant 6$) has been computed, and for each fatgraph the isomorphism group is known. The full list of fatgraphs and their isomorphisms is too long to reproduce here (see a sample in Appendix C), but the data is publicly available at http://fatghol.googlecode.com/download/list. A summary of the number of abstract and marked fatgraphs is provided in Tables 3.2 and 3.3 in Chapter 3.

3. A novel parallel algorithm for exact Gaussian Elimination of general sparse matrices

The algorithms presented in Chapter 3 reduce computation of the Betti number of moduli spaces $\mathcal{M}_{g,n}$ to reckoning the rank (over $\mathbb{Q}$) of some large sparse matrix with integer entries. An effective method for computing this rank is given by Gaussian Elimination.

The serial algorithm for Gaussian Elimination is well-known; it consists of a certain number of iterations of the following two steps: a *pivoting* step followed by an *elimination* step. Starting with the upper left entry, a non-zero element (pivot)

is searched for; once a pivot has been found, a permutation is applied so that the pivot rests in the upper left corner of the "uneliminated" matrix. In the elimination phase, all the elements in the leftmost column and below the pivot are set to zero by summing to each row a suitable multiple of the pivot row. Then the procedure is recursively applied to the portion of the matrix excluding the topmost row and the leftmost column.

An enormous literature has been published on the subject of Gaussian Elimination, that is outside the scope of this short introduction to survey. However, available practical codes for exact matrix rank computation seem to be limited to the free software library LinBox [14, 43], which does not offer any parallel distributed-memory algorithm and is thus bound to usage on a single computer at a time. Indeed, Gaussian Elimination algorithms with exact computations has been analysed in [16], and the authors concluded that "there remains to design a direct parallel method suited to sparse matrices".

Let us say that a matrix is in "block echelon form" iff each row starts with a number of null entries that is *not less* than the number of null entries of the row directly above. The "Rheinfall" algorithm presented here is based on the observation that any sparse matrix can be put in a "block echelon form" with minimal computational effort. One can then run elimination on each block of rows of the same length independently (i.e., in parallel); a communication step is needed to re-order the rows after elimination. The procedure ends when all blocks have been reduced to a single row, i.e., the matrix has been put in row echelon form.

As can be seen from this cursory description, the "Rheinfall" algorithm distributes the matrix data and the elimination work to an arbitrary number p of processors; it can thus fully exploit the power of present-day massively parallel machines. No collective communication takes place; however it is assumed that the communication fabric is able to route a message of arbitrary size[2] from any processor to any other. On the other hand, the algorithm requires frequent one-to-one communication among all processing units; the issue of distributing matrix data among processors is thus crucial to minimizing the network traffic and for the overall performance of the algorithm.

The "Rheinfall" algorithm has been developed and used to compute the rank of the homology matrices arising from fatgraph complexes; however, it is of (potentially) much wider application. The interesting question is then to determine the class of matrices on which "Rheinfall" is efficient. A direct approach to this problem quickly leads to the conclusion that the amount of work and communication performed by the algorithm is strictly dependent on the nonzero pattern of the input matrix. By using computational experiments, we can assess the performance of "Rheinfall" on a large set of matrices, and give some complexity estimates. As it turns out, "Rheinfall" is competitive with LinBox on a large subset of the test matrices, and specifically outperforms it on the $\mathcal{M}_{g,n}$ homology ones.

4. Notation

Algorithms are listed in pseudo-code reminiscent of the Python language syntax (see [65]); comments in the code listings are printed in *italics* font. The word "object" is used to denote a kind of aggregate type in the computer programs: an *object* is a tuple '$(a_1, a_2, \ldots, a_N)$', where each of the slots a_i can be independently assigned

[2]Actually, the maximum size of a message is a linear function of the length of a matrix row.

a value; we write $X.a_i$ to denote the slot a_i of object X. Object slots are mutable, i.e., they can be assigned different values over the course of time. Appendix A gives a complete recap of the notation used and the properties assumed of syntax, data structures, and operators.

A great deal of Chapter 3 is concerned with finding computationally-effective representations of topological objects; in general, we use boldface letters to denote the computer analog of a mathematical object. For instance, the letter G always denotes a fatgraph, and $\boldsymbol{G}$ its corresponding computer representation as a *Fatgraph* object.

Finally, if A is a category of which X, Y are objects, we use Eilenberg's notation $A(X, Y)$ for the Hom-set, instead of the more verbose $\mathrm{Hom}_A(X, Y)$.

CHAPTER 2

The Fatgraph Chain Complex

This chapter recalls the main definitions and properties of fatgraphs, and the relation of the fatgraph complex to the cohomology of $\mathcal{M}_{g,n}$. These results are well-known: a comprehensive exposition of the research connected with the topic of this chapter can be found in Mondello's [51]; the book by Lando and Zvonkin [42] provides a broad survey of the applications of fatgraphs and an introduction accessible to readers without a background in Algebraic Geometry; a recent account of the crucial Jenkins-Strebel theorem (together with applications to the triangulations of the Teichmüller and moduli space of curves) can be read in Arbarello and Cornalba's paper [3].

1. Fatgraphs

"Fatgraphs" take their name from being usually depicted as graphs with thin bands as edges, instead of 1-dimensional lines; they have also been called "ribbon graphs" in Algebraic Geometry literature. Here, the two names will be used interchangeably.

Definition 2.1 (Topological definition of fatgraphs)**.** A fatgraph is a finite CW-complex of pure dimension 1, together with an assignment, for each vertex v, of a cyclic ordering of the edges incident at v.

A morphism of fatgraphs is a cellular map $f\colon G \to G'$ such that, for each vertex v of G', the preimage $f^{-1}(V)$ of a small neighborhood V of v is a small neighborhood of a tree in G (i.e., $f^{-1}(V)$ is a contractible connected graph).

Unless otherwise specified, we assume that all vertices of a fatgraph have valence at least 3.

If G is a fatgraph, denote $V(G)$, $E(G)$ and $L(G)$ the sets of vertices, unoriented edges and oriented edges (equivalently called "legs" or "half-edges").

1.1. Combinatorial description of fatgraphs.

The following combinatorial description of a fatgraph will also be needed:

Definition 2.2 (Combinatorial definition of fatgraphs)**.** A fatgraph is a 4-tuple $(L, \sigma_0, \sigma_1, \sigma_2)$, comprised of a finite set L together with bijective maps $\sigma_0, \sigma_1, \sigma_2\colon L \to L$ such that:

- σ_1 is a fixed-point free involution: $\sigma_1^2 = \mathrm{id}$, and
- $\sigma_0 \circ \sigma_2 = \sigma_1$.

Lemma 2.3. *Definitions 2.1 and 2.2 are equivalent.*

PROOF. To pass from the topological description to the combinatorial one, take L to be the set of *oriented* edges of the CW-complex underlying a fatgraph. Define $\sigma_1 \colon L \to L$ as the orientation reversal on edges. Define $\sigma_0 \colon L \to L$ by means of the cyclic order at vertices: let $L(v)$ be the subset of edges in L that *end* at a vertex v, the cyclic order on edges incident at v induces a cyclic order on $L(v)$. If $x \in L(v)$ then define $\sigma_0(x)$ as the successor to x in the cyclic order on $L(v)$. Finally, define $\sigma_2 : L \to L$ by means of $\sigma_2 = \sigma_0^{-1}\sigma_1$.

Vice versa, let L_i be the set of orbits of the map σ_i. Take L_0 to be the set of 0-cells; for each $\{x^+, x^-\} \in L_1$, glue a 1-cell to the 0-cells corresponding to the σ_0-orbits of x^+ and x^-. The cyclic order at each vertex is induced by the action of σ_0. □

Any two of the maps $\sigma_0, \sigma_1, \sigma_2$ determine the third, by means of the defining relation $\sigma_0 \circ \sigma_2 = \sigma_1$; therefore, to give a ribbon graph, it is sufficient to specify only two out of three maps.

In the combinatorial description, $V(G)$ is the set L_0 of orbits of σ_0, $E(G)$ is the set L_1 of orbits of σ_1, and $L(G)$ is plainly the set L.

1.2. Morphisms arising from contraction of an edge. Let G be a fatgraph, and G' be the CW-complex obtained by contracting an edge $x \in E(G)$ to a point. If x connects two *distinct* vertices (i.e., x is not a loop) then G' inherits a fatgraph structure from G: if $(x < x_1 < \ldots < x_k < x)$ and $(x < x'_1 < ... < x'_h < x)$ are the cyclic orders at endpoints of x, then the vertex formed by collapsing x is endowed with the cyclic order $(x_1 < \ldots < x_k < x'_1 < \ldots < x'_h)$.

Contraction morphisms play a major role in manipulation of ribbon graphs.

Lemma 2.4. *Any morphism of fatgraphs is a composition of isomorphisms and contractions of non-loop edges.*

We can thus define functors $V(-)$, $E(-)$ and $L(-)$ that take a morphism of graphs to a map of their set of vertices, (unoriented) edges, and oriented edges.

1.3. From fatgraphs to Riemann surfaces. There is a functorial construction to build a closed oriented surface $S(G)$ from a fatgraph G; this is usually referred to as "thickening" or "fattening" in the literature.

Lemma 2.5. *(1) There exists a functor S that associates to every fatgraph G a punctured Riemann surface $S(G)$, and to every morphism $f \colon G \to G'$ a continuous map $S(f) \colon S(G) \to S(G')$. (2) The surface $S(G)$ is naturally endowed with a triangulation indexed by oriented edges of G. (3) The graph G is a deformation retract of $S(G)$.*

Denote by $B(G)$ the set L_2 of orbits of σ_2; in the topological description, its elements are the support of 1-cycles in $H^1(G)$ that correspond under the retraction to small loops around the punctures in $S(G)$; they are called "boundary cycles" of G.

The assignment $G \mapsto B(G)$ extends to a functor $B(-)$; by Lemma 2.4, for any $f \colon G_1 \to G_2$ the map $B(f) \colon B(G_1) \to B(G_2)$ is a bijection.

The correspondence between fatgraphs and Riemann surfaces allows us to give the following.

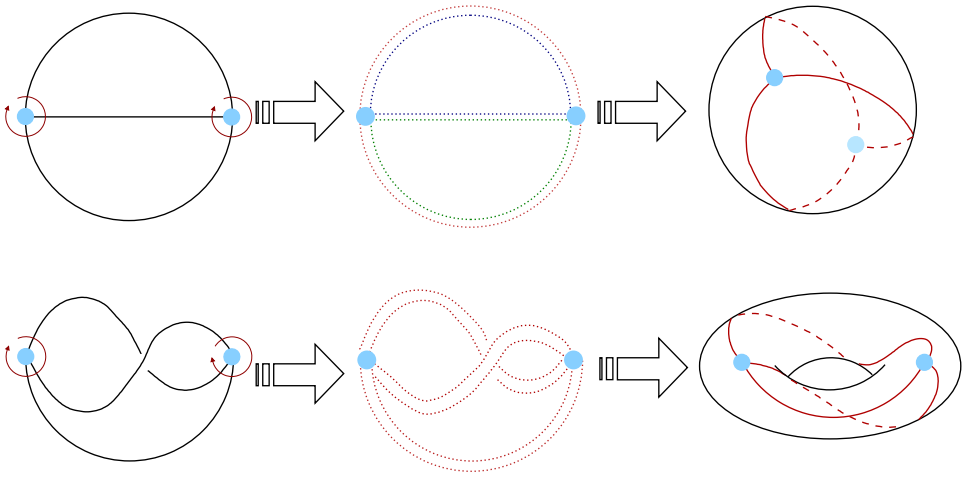


FIGURE 2.1. Thickening of a fatgraph into a Riemann surface. *Left column:* Starting fatgraph: the cyclic order at the vertices is given by the orientation of the ambient euclidean plane. *Middle column:* Thickening of the fatgraph by gluing topological disks along the boundary components. The border of a cells is drawn as a dotted line; each topological disk has been given a different color. *Right column:* The resulting Riemann surface with the embedded graph. Note that the two starting graphs would be isomorphic when considered as ordinary multigraphs; they are distinguished by the additional cyclic structure at the vertices.

Definition 2.6. The *number of boundary cycles* of a graph G is given by $n = |B(G)|$, and is equal to the puncture number of the Riemann surface $S(G)$.

If $S(G)$ has genus g and n punctures, then:

$$\chi(G) = \chi(S(G)) = 2 - 2g - n = 2 - 2g - |B(G)|, \tag{1.1}$$

so we can define, for any fatgraph G, the *genus* g, as given by the above relation.

It is trivial to check the following.

Lemma 2.7. *If G' is obtained from G by contraction of a non-loop edge, then G and G' share the same genus and number of boundary cycles.*

1.4. Complex analytic structure on $S(G)$. We can give the topological Riemann surface $S(G)$ a complex analytic structure by means of the triangulation in Lemma 2.5*(2)* and an analytic atlas, which depends on the perimeters $p_1, \ldots, p_n$ of boundary cycles $b_1, \ldots, b_n$. (See [52] for details.)

Definition 2.8. A metric ℓ on a fatgraph G is an assignment of a real positive number ℓ_x for each edge $x \in E(G)$.

A metrized fatgraph (G, ℓ) is a fatgraph G equipped with a metric ℓ.

The perimeters $p_1, \ldots, p_n$ are determined by the metric data ℓ, therefore the complex analytic structure on $S(G)$ actually depends on the metrized graph (G, ℓ). Let $S(G, \ell)$ denote the Riemann surface $S(G)$ endowed with this complex analytic structure.

Lemma 2.9. *If G, G' are metrized fatgraphs and f is an isomorphism preserving edge lengths, then $S(f)$ is a complex analytic mapping.*

1.5. Marked fatgraphs.

Definition 2.10. A *marked fatgraph* is a fatgraph G endowed with a bijection $\nu\colon B(G) \to \{1,\dots,n\}$. The map ν is called the "marking" on G.

A morphism $f\colon G_1 \to G_2$ of marked fatgraphs must preserve the numbering of boundary cycles:

$$\begin{array}{ccc} B(G_1) & \xrightarrow{\quad f \quad} & B(G_2) \\ & {\scriptstyle \nu_1}\searrow \quad \swarrow{\scriptstyle \nu_2} & \\ & \{1,\dots,n\} & \end{array} \tag{1.2}$$

By a slight abuse of language, we shall usually omit mention of the marking map ν and just speak of "the marked fatgraph G".

2. Moduli spaces of Riemann surfaces

Let us recap the main points of the construction of the moduli space of smooth algebraic curves; the short summary given here tracks closely the first section of [45], which has proofs and references.

Fix integers $g, n \geqslant 0$ such that $2-2g-n < 0$. Let S be a Riemann surface of genus g and $P = \{P_1,\dots,P_n\}$ a set of n points in S.

Let $\mathrm{Diff}(S,P)$ be the group of diffeomorphisms of S that fix P pointwise; $\mathrm{Diff}^0(S,P)$ denotes the subgroup of diffeomorphisms homotopic to the identity mapping id_S; let $\mathrm{Diff}^+(S,P)$ indicate the subgroup of orientation-preserving diffeomorphisms.

Every set P of marked points can be transformed into another chosen set P' (of the same cardinality) by a diffeomorphism ϕ homotopic to the identity mapping id_S. Therefore, $\mathrm{Diff}^+(S,P)$ and $\mathrm{Diff}^0(S,P)$ depend only on $n = |P|$ and not on P (see [41]). Summing up:

Definition 2.11. $\mathrm{Diff}(S,n)$ is the group of orientation-preserving diffeomorphisms that keep the n marked points fixed.

$\mathrm{Diff}^0(S,n)$ is the group of diffeomorphisms of S which are homotopic to the identity mapping id_S and that keep the n marked points fixed.

Every complex structure on S determines a conformal structure; let $\mathrm{Conf}(S)$ be the set of all conformal structures on S.

Definition 2.12. The Teichmüller space

$$\mathcal{T}_{g,n} := \mathrm{Conf}(S)/\mathrm{Diff}^0(S,n)$$

is the quotient of the set of all conformal metrics on S by the set of all diffeomorphisms homotopic to the identity and fixing the n marked points.

The Teichmüller space $\mathcal{T}_{g,n}$ is an analytic space and is homeomorphic to a convex domain in $\mathbb{C}^{3g-3+n}$.

Definition 2.13. The mapping class group $\Gamma_{g,n}$ is the set of connected components of $\mathrm{Diff}^+(S,n)$, the group of all diffeomorphisms that preserve orientation and fix marked points:

$$\Gamma_{g,n} := \mathrm{Diff}^+(S,n)/\mathrm{Diff}^0(S,n).$$

Definition 2.14. The topological space $\mathcal{M}_{g,n} := \mathcal{T}_{g,n}/\Gamma_{g,n}$ is the moduli space of (smooth) n-pointed algebraic curves of genus g. It parametrizes complex structures on S, up to diffeomorphisms that: *(1)* are homotopic to the identity mapping on S, *(2)* preserve the orientation of S, and *(3)* fix the n marked points.

Since $\mathcal{T}_{g,n}$ is an analytic variety and $\Gamma_{g,n}$ acts discontinuously with finite stabilizers, $\mathcal{M}_{g,n}$ inherits a structure of analytic orbifold of complex dimension $3g-3+n$.

Since $\mathcal{T}_{g,n}$ is contractible, its equivariant (co)homology with rational coefficients is isomorphic to the rational (co)homology of $\mathcal{M}_{g,n}$ (see [10, VII.7.7]):

$$H_*^{\Gamma_{g,n}}(\mathcal{T}_{g,n},\mathbb{Q}) \cong H_*(\mathcal{M}_{g,n},\mathbb{Q}), \qquad H^*_{\Gamma_{g,n}}(\mathcal{T}_{g,n},\mathbb{Q}) \cong H^*(\mathcal{M}_{g,n},\mathbb{Q}), \tag{2.1}$$

One may instead consider equivalence classes of n-punctured surfaces S (i.e., with n points *removed*) by bianalytic mappings that do *not* permute the punctures, and repeat the same construction of the Teichmüller and the moduli space. By the Riemann extension theorem, the two approaches turn out to yield the same result.

In the course of this chapter, we will make use of the description of $\mathcal{M}_{g,n}$ that best fits in the context, often without explicit notice. In particular, we shall consider the points $P_1, \ldots, P_n$ as marked points or as punctures, interchangeably.

2.1. Quadratic differentials. If S is an n-punctured Riemann surface, then S retracts onto a graph, however, this graph is not uniquely determined. We can refine this correspondence: a theorem proved independently by J. A. Jenkins [37] and K. Strebel [59] provides the key tool: the construction of fatgraphs from smooth complex curves.

Definition 2.15. A quadratic differential q on a Riemann surface S is a (meromorphic) section of $(T^*S)^{\otimes 2}$.

The set of vectors in T_zS on which q takes real non-negative values forms a real line in T_zS: therefore, they make up a foliation F on $S\setminus\{\text{poles of } q\}$. The non-compact leaves of F together with zeroes of q form the "critical locus" of q. Call F the "horizontal" foliation associated with q.

Every quadratic differential q induces a metric (away from the critical locus) by $\mathrm{d}s^2 = |q(z)|\cdot|\mathrm{d}z|$.

Theorem 2.16 (Jenkins, Strebel; [59, Theorem 23.2 and 23.5])**.** *For any complex analytic curve S with n marked points $P_1,\ldots,P_n$, and any assignment of real positive numbers $p_1,\ldots,p_n$, there exists one and only one quadratic differential q such that:*

- *q is holomorphic on $S\setminus\{P_1,\ldots,P_n\}$;*
- *q has double poles at the marked points $P_1,\ldots,P_n$ with second residue $-(p_1/2\pi)^2, \ldots, -(p_n/2\pi)^2$;*
- *the non-critical real trajectories of F_q are simple closed circles around x_i;*
- *the complement of the critical locus is a collection of disks $\{D_i\}_{i=1,\ldots,n}$, each one centered at a pole P_i.*

Furthermore, q has the following properties:

- *every nonsingular closed leaf circling around P_i has length p_i in the flat metric induced by q.*

» *the critical locus G of q is a graph embedded in S;*
» *the projective class of the collection of radii of disks $\{D_i\}$ equals the projective class $[p_1, \dots, p_n]$;*
» *q depends continuously on S and $(p_1, \dots, p_n)$.*

2.2. The fatgraph cellularization of the moduli spaces of marked Riemann surfaces. An embedding of a fatgraph G is an injective continuous map $\iota\colon G \to S$, that is, a homeomorphism of G onto $\iota(G) \subseteq S$, such that the orientation on S induces the cyclic order at the vertices of $\iota(G)$.

Definition 2.17. An *embedded fatgraph* is a fatgraph G endowed with a homeomorphism $\tilde{\iota}$ between $S(G)$ and the ambient surface S, modulo the action of $\mathrm{Diff}^0(S)$.

There is an obvious action of $\Gamma_{g,n}$ on the set $\widetilde{\mathcal{R}}_{g,n}$ of fatgraphs embedded into n-marked Riemann surfaces of genus g.

If confusion is likely to arise, we shall speak of *abstract* fatgraphs, to mean the topological and combinatorial objects defined in Definitions 2.1 and Definition 2.2, as opposed to *embedded* fatgraphs as in Definition 2.17 above.

The critical graph G inherits a structure of embedded metrized fatgraph from the ambient surface S: the length of an edge x is the one measured in the metric induced by the quadratic differential. Furthermore, Jenkins-Strebel's theory states that G has a vertex of valence $k+2$ where q has a zero of order k, therefore, vertices of G have valence $\geqslant 3$. Since the markings $P_1, \dots, P_n$ are *ordered*, G has an additional structure of *marked* fatgraph.

Let G be a fatgraph (embedded or abstract) of genus g with n marked boundary components. The set $\Delta(G) = \{(G, \ell)\}$ of metrics on G has an obvious structure of topological cell; now glue these cells by stipulating that $\Delta(G')$ is the face $\ell_x = 0$ of $\Delta(G)$ when G' is obtained from G by contraction of the edge x. The topological spaces obtained by these gluing instructions are denoted $\mathcal{T}^{\mathrm{comb}}_{g,n}$ (when using *embedded* fatgraphs), or $\mathcal{M}^{\mathrm{comb}}_{g,n}$ (when using *abstract* fatgraphs). The following theorem clarifies their relation to the Teichmüller and the moduli space; details can be found, e.g., in [**51**, Section 4.1].

Theorem 2.18. *The thickening construction induces orbifold isomorphisms:*

$$\mathcal{T}_{g,n} \times \mathbb{R}^n \simeq \mathcal{T}^{\mathrm{comb}}_{g,n}, \qquad \mathcal{M}_{g,n} \times \mathbb{R}^n \simeq \mathcal{M}^{\mathrm{comb}}_{g,n},$$

Call $M(G)$ the cell in $\mathcal{M}^{\mathrm{comb}}_{g,n}$ corresponding to an abstract fatgraph G, and $T(\tilde{G})$ the cell in $\mathcal{T}^{\mathrm{comb}}_{g,n}$ corresponding to an embedded fatgraph $\tilde{G}$.

The functorial action of $\Gamma_{g,n}$ on $\widetilde{\mathcal{R}}_{g,n}$ induces an action on $\mathcal{T}^{\mathrm{comb}}_{g,n}$, which permutes cells $T(\tilde{G})$ by PL isomorphisms.

Lemma 2.19. *$\mathcal{M}^{\mathrm{comb}}_{g,n}$ is the quotient space of $\mathcal{T}^{\mathrm{comb}}_{g,n}$ by the cellular action of the mapping class group $\Gamma_{g,n}$; the projection homomorphism commutes with the isomorphisms in Theorem 2.18.*

The action of $\Gamma_{g,n}$ commutes with the face operators, so $M(G)$ is a face of $M(G')$ iff G' is obtained from G by contraction of a non-loop edge.

Lemma 2.20. *The isotropy group $\Gamma_{\tilde{G}}$ of the cell $T(\tilde{G}) \hookrightarrow \mathcal{T}^{\mathrm{comb}}_{g,n}$ is (isomorphic to) the automorphism group $\mathrm{Aut}\, G$ of the abstract fatgraph G underlying $\tilde{G}$.*

PROOF. If $a \in \operatorname{Aut} G$, then $S(a)$ is an automorphism of $S(G, \ell)$ for any metric ℓ. If $\ell \in \Delta(G)$ varies continuously, then so does $S(a)$. Therefore, an element of $\Gamma_{\tilde{G}} \subseteq \Gamma_{g,n}$ is defined.

Conversely, let $\tau \in \Gamma_{g,n}$ fix the cell $T\tilde{G}$ *setwise*. If q is the Jenkins-Strebel quadratic differential inducing the complex analytic structure corresponding to the metric $\ell \in T(\tilde{G})$, then $\tau^* q$ defines a quadratic differential corresponding to a point in $T\tilde{G}$, so it has critical graph $\tilde{G}$. But $\tau^* q$ has critical graph $\tau(\tilde{G})$, since q has critical graph $\tilde{G}$. Therefore, τ restricts to a fatgraph isomorphism, so a map $\Gamma_{\tilde{G}} \to \operatorname{Aut} G$ is defined, which is clearly the inverse of the map $\operatorname{Aut} G \to \Gamma_{\tilde{G}}$ defined above. □

3. Equivariant homology of $\mathcal{T}_{g,n}$ and the complex of fatgraphs

Definition 2.21. An orientation of a fatgraph G is an orientation of the vector space $\mathbb{Q}E(G)$, that is, the choice of an order of the edges of G, up to even permutations.

Giving an orientation on G is the same as orienting the simplex $\Delta(G)$; respectively, an orientation on $\tilde{G}$ is identified with an orientation of the cell $T(\tilde{G})$.

If G is a fatgraph with p edges, let $W_G := \bigwedge^p \mathbb{Q}E(G)$ be the 1-dimensional vector space generated by the wedge products $x_1 \wedge \ldots \wedge x_p$ of edges of G. Every $f \in \operatorname{Aut} G$ induces a map $f : E(G) \to E(G)$ on the edges and thus a map $f_* : x_1 \wedge \ldots \wedge x_p \mapsto f(x_1) \wedge \ldots \wedge f(x_p)$. Trivially, $f_*(x_1 \wedge \ldots \wedge x_p) = \pm x_1 \wedge \ldots \wedge x_p$, depending on whether f preserves or reverses the orientation of G.

Definition 2.22. A fatgraph G is *orientable* iff it has no orientation-reversing automorphisms.

Form a differential complex of orientable fatgraphs as follows.

Definition 2.23. The complex (W_*, D) of orientable fatgraphs is defined by:

- $W_p := \bigoplus_G W_G$, where G runs over orientable fatgraphs with $(2g+n-1+p)$ edges;
- $D := \sum_1^p (-1)^i d_i$, where $d_i \colon W_p \to W_{p-1}$ is given by:

$$d_i(x_1 \wedge \ldots \wedge x_p) := \begin{cases} x_1 \wedge \ldots \wedge \widehat{x_i} \wedge \ldots \wedge x_p & \text{if } x_i \text{ is not a loop and} \\ & G/x_i \text{ is orientable,} \\ 0 & \text{otherwise.} \end{cases}$$

Call a fatgraph with one vertex only a *clover*; the number of edges of a clover of genus g with n boundary cycles is readily computed by (1.1):

$$m_{\min} = 1 - \chi(G) = 2g + n - 1. \tag{3.1}$$

On the other hand, the number m of edges and the number l of vertices are maximal when all vertices are 3-valent:

$$m_{\max} = 6g + 3n - 6, \qquad l_{\max} = 4g + 4n - 4 \tag{3.2}$$

From equations (3.1) and (3.2) we see that W_* is a finite complex of length $4g + 2n - 5$, which is already predicted by the results of Harer on the equivariant spine of $\mathcal{T}_{g,n}$ [30].

Every oriented fatgraph (G, ω) defines an element $\omega_G \in W_G$ by taking the wedge product of edges of G in the order given by ω; conversely, any $x_1 \wedge \ldots \wedge x_p \in W_G$ defines an orientation on G by setting $\omega := x_1 < \ldots < x_p$.

Theorem 2.24. *The $\Gamma_{g,n}$-equivariant homology of $\mathcal{T}_{g,n}$ with rational coefficients is computed by the complex of oriented fatgraphs (W_*, D), i.e., there exists an isomorphism:*

$$H_*^{\Gamma_{g,n}}(\mathcal{T}_{g,n}, \mathbb{Q}) \cong H_*(W_*, D).$$

Proof. The genus and number of boundary cycles will be fixed throughout, so for brevity, set $\Gamma := \Gamma_{g,n}$, $\mathcal{T} := \mathcal{T}_{g,n}$ and $\mathcal{T}^{\mathrm{comb}} := \mathcal{T}^{\mathrm{comb}}_{g,n}$.

By Theorem 2.18, we have:

$$H_*^{\Gamma}(\mathcal{T}, \mathbb{Q}) = H_*^{\Gamma}(\mathcal{T}^{\mathrm{comb}}, \mathbb{Q}).$$

Recall that $H_*^{\Gamma}(\mathcal{T}^{\mathrm{comb}}, \mathbb{Q})$ can be defined as the homology of the double complex $P_* \otimes C_*(\mathcal{T}^{\mathrm{comb}}, \mathbb{Q})$, where P_* is any projective resolution of $\mathbb{Q}$ over $\mathbb{Q}[\Gamma]$. The spectral sequence $E^1_{pq} := H_q(P_* \otimes_\Gamma C_p) = H_q(\Gamma, C_p)$ abuts to $H^{\Gamma}_{p+q}(\mathcal{T}^{\mathrm{comb}})$ (see [10, VII.5 and VII.7]).

The space $\mathcal{T}^{\mathrm{comb}}_{g,n}$ has, by definition, an equivariant cellularization with cells indexed by embedded fatgraphs of genus g with n marked boundary components. Let X_p be a set of representatives for the orbits of p-cells under the action of Γ. By Lemma 2.19, X_p is in bijective correspondence with the set of abstract fatgraphs having p edges, and the orientation of a cell translates directly to an orientation of the corresponding graph. For each geometric simplex $T(\tilde{G}) \subseteq \mathcal{T}^{\mathrm{comb}}$, let $\Gamma_{\tilde{G}}$ be its isotropy group, and let $\mathbb{Q}\tilde{G}$ be the $\Gamma_{\tilde{G}}$-module consisting of the $\mathbb{Q}$-vector space generated by an element Δ on which $\Gamma_{\tilde{G}}$ acts by the orientation character: $\tau \cdot \Delta = \pm\Delta$ depending on whether τ preserves or reverses the orientation of the cell $T(\tilde{G})$. By Lemma 2.20, there is an isomorphism between $\Gamma_{\tilde{G}}$ and $\operatorname{Aut} G$; if $\tau \in \Gamma_{\tilde{G}}$ reverses (resp. preserves) orientation of $T(\tilde{G})$, then the corresponding $f \in \operatorname{Aut} G$ reverses (resp. preserves) orientation on G. Therefore, $\mathbb{Q}\tilde{G}$ and W_G are isomorphic as $\operatorname{Aut} G = \Gamma_{T\tilde{G}}$ modules.

Following [10, p. 173], let us decompose (as a Γ-module)

$$C_*(\mathcal{T}^{\mathrm{comb}}, \mathbb{Q}) = \bigoplus_{G \in X_p} W_G;$$

then, by Shapiro's lemma [10, III.6.2], we have:

$$H_q(\Gamma, C_p) \cong \bigoplus_{G \in X_p} H_q(\Gamma_{\tilde{G}}, \mathbb{Q}\tilde{G}) \cong \bigoplus_{G \in X_p} H_q(\operatorname{Aut} G, W_G).$$

Since $\operatorname{Aut} G$ is finite and we take rational coefficients, then $H_q(\operatorname{Aut} G, W_G) = 0$ if $q > 0$ [10, III.10.2]. On the other hand, if G is orientable then $\operatorname{Aut} G$ acts trivially on W_G, so:

$$H_0(\operatorname{Aut} G, W_G) = \begin{cases} 0 & \text{if } G \text{ has an orientation-reversing automorphism,} \\ W_G & \text{if } G \text{ has } \textit{no} \text{ orientation-reversing automorphisms.} \end{cases}$$

Let X'_p be the collection of all *orientable* fatgraphs with p edges. Substituting back into the spectral sequence, we see that only one column survives:

$$E^1_{p,0} = \bigoplus_{G \in X'_p} W_G = W_p, \tag{3.3}$$

$$E^1_{p,q} = 0 \qquad \text{for all } q > 0, \tag{3.4}$$

In other words, E^1_{pq} reduces to the complex $(E^1_{*,0}, d^1)$.

Finally, we show that the differential $d^1 \colon E^1_{p,0} \to E^1_{p-1,0}$ corresponds to the differential $D \colon W_p \to W_{p-1}$ under the isomorphism formula (3.3); this will end the

proof. Indeed, we shall prove commutativity of the following diagram at the chain level:

$$\begin{array}{ccccccc} P_*\otimes W_p & = & \bigoplus_{G\in X'_p} P_*\otimes W_G & \xrightarrow{\mathrm{id}_P\otimes D} & \bigoplus_{G'\in X'_{p-1}} P_*\otimes W_{G'} & = & P_*\otimes W_{p-1} \\ & & \downarrow{\scriptstyle \theta_p} & & \downarrow{\scriptstyle \theta_{p-1}} & & \\ & & P_*\otimes C_p(\mathcal{T}^{\mathrm{comb}},\mathbb{Q}) & \xrightarrow[\mathrm{id}_P\otimes\partial]{} & P_*\otimes C_{p-1}(\mathcal{T}^{\mathrm{comb}},\mathbb{Q}) & & \end{array} \tag{3.5}$$

which implies commutativity at the homology level:

$$\begin{array}{ccc} \bigoplus_{G\in X'_p} H_0(\mathrm{Aut}\, G, W_G) & \xrightarrow{D} & \bigoplus_{G'\in X'_{p-1}} H_0(\mathrm{Aut}\, G', W_{G'}) \\ \downarrow{\scriptstyle \cong} & & \downarrow{\scriptstyle \cong} \\ H_0(\Gamma, C_p(\mathcal{T}^{\mathrm{comb}},\mathbb{Q})) & \xrightarrow[d^1=H_0(\Gamma,\partial)]{} & H_0(\Gamma, C_{p-1}(\mathcal{T}^{\mathrm{comb}},\mathbb{Q})) \end{array}$$

whence the conclusion $E^1_{*,0}\cong (W_*, D)$.

The vertical maps θ_p, θ_{p-1} in (3.5) are the chain isomorphisms underlying the Γ-module decomposition $C_p(\mathcal{T}^{\mathrm{comb}},\mathbb{Q})\cong\bigoplus_{G\in X_p} W_G$. Taking the boundary of a cell $T(\tilde{G})\subseteq\mathcal{T}^{\mathrm{comb}}$ commutes with the Γ-action: $\partial T(\tau\cdot\tilde{G})=\tau\cdot\partial T(\tilde{G})$. Furthermore, $T(\tilde{G}')$ is a cell in $\partial T(\tilde{G})$ iff $\tilde{G}'$ is obtained from $\tilde{G}$ by contraction of an edge; but $\tilde{G}'$ is a contraction of $\tilde{G}$ iff the underlying *abstract* fatgraphs G' and G stand in the same relation. Thus, the Γ-complexes (C_*,∂) and (W_*, D) are isomorphic by θ_*, so diagram (3.5) commutes, as was to be proved. □

CHAPTER 3

Algorithms for graph homology

The objects of this Chapter 3 are finding an effectively computable representation of fatgraphs (see Section 1), and presenting algorithms to:

(1) compute automorphisms of any given fatgraph (Section 2);
(2) generate the set $\mathcal{R}_{g,n}$ of fatgraphs, given the genus g and number of boundary components n (Section 3);
(3) compute the homology of the fatgraph complex $\mathcal{R}_{g,n}$ (Section 4).

By Theorem 2.24, this is tantamount to computing the (co)homology with rational coefficients of the moduli spaces $\mathcal{M}_{g,n}$.

An effective computer implementation[1] of the algorithm has been written as part of the research. It is capable of computing the Betti numbers of $\mathcal{M}_{g,n}$ for $(2g+n) \leqslant 6$ on commonly-available hardware. Experimental results from running the code are discussed in Section 5.

1. Computer representation of Fatgraphs

Although the combinatorial definition of a fatgraph (cf. Lemma 2.3) lends itself to a computer representation as a triple of permutations —as used, e.g., in [57, Section 2.4]—, the functions that are needed by the generation algorithms (see Section 3) are rather topological in nature and thus suggest an approach more directly related to the concrete realization of a fatgraph.

Definition 3.1. A *Fatgraph* object $\boldsymbol{G}$ is comprised of the following data:

» A list $\boldsymbol{G}$.*vertices* of *Vertex* objects.
» A list $\boldsymbol{G}$.*edges* of *Edge* objects.
» A set $\boldsymbol{G}$.*boundary_cycles* of *BoundaryCycle* objects.
» An orientation $\boldsymbol{G}$.*orient*.

The exact definition of the constituents of a *Fatgraph* object is the subject of the following sections; informally, let us say that a *Vertex* is a cyclic list of edges and that an *Edge* is a pair of vertices and incidence positions. A precise statement about the correspondence of abstract fatgraphs and *Fatgraph* objects is made in Section 1.5.

There is some redundancy in the data comprising a *Fatgraph* object: some of these data are inter-dependent and cannot be specified arbitrarily. Actually, all data comprising a *Fatgraph* object can be computed from the vertex list alone, as the following sections show.

In what follows, the letters l, m and n shall denote the number of vertices, edges and boundary cycles respectively:

[1] Code available for download from http://code.google.com/p/fatghol.

» $l = |V(G)| = \mathit{size}(\boldsymbol{G}.\mathit{vertices})$,
» $m = |E(G)| = \mathit{size}(\boldsymbol{G}.\mathit{edges})$,
» $n = |B(G)| = \mathit{size}(\boldsymbol{G}.\mathit{boundary\_cycles})$.

Throughout this Chapter, we shall use the topological and the combinatorial definition of a fatgraph equivalently, according to what best suits in context. The symbols σ_0, σ_1 and σ_2 stand for the structure maps in the combinatorial definition Definition 2.2.

For integers α and k, we use $(\alpha\%k)$ to denote the smallest non-negative representative of $\alpha \mod k$.

1.1. Vertices. We can represent a fatgraph vertex by assigning labels[2] to all fatgraph vertices and mapping a vertex to the cyclically-invariant list of labels of incident edges. Figure 3.1 gives an illustration.

Definition 3.2. A vertex together with a choice of an attached edge is called a *ciliated* vertex. The chosen edge is called the *cilium*.

Definition 3.3. If v is a ciliated vertex and e is a half-edge attached to it, define the *attachment index* of e at v as the index of edge e relative to the cilium at v: if α is the attachment index of e at v, then σ_0^α takes the cilium at v onto e.

The attachment index at a vertex is unambiguously defined for all edges which are not loops; the two half-edges comprising a loop have distinct attachment indices. For brevity, in the following we shall slightly abuse the definition and speak of *the* attachment index of an edge at a vertex.

Definition 3.4. A *Vertex* object $\boldsymbol{v} = \mathit{Vertex}(e_1, \ldots, e_z)$ is a list of the labels e_1, $\ldots$, e_z of attached edges.

Two *Vertex* objects are considered equal if one is equal (as a sequence) to the other rotated by a certain amount.

Example 3.5. Consider for example the fatgraph depicted in Figure 3.1: the edges are given labels "0", "1" and "2", so the two vertices are represented by objects $\boldsymbol{a} = \mathit{Vertex}(0,1,2)$ and $\boldsymbol{b} = \mathit{Vertex}(0,2,1)$, where the edge labels are listed starting with the ciliated vertex and continue according to the cyclic order given at the vertex.

Note that the definition of *Vertex* objects as plain lists would correspond to *ciliated* vertices in a fatgraph. For instance, if a different cilium were chosen in Figure 3.1, we would have represented vertex $\boldsymbol{a}$ equivalently as $\mathit{Vertex}(1,2,0)$ or $\mathit{Vertex}(2,0,1)$. In order to implement the cyclic behavior of fatgraph vertices, the requirement on equality must be imposed; equality of *Vertex* objects can be tested by an algorithm of quadratic complexity in the vertex valence.

If $\boldsymbol{v}$ is a vertex object, let us denote $\mathit{num\_loops}(\boldsymbol{v})$ the number of loops attached to $\boldsymbol{v}$. This is a vertex invariant and will be used in the computation of fatgraph isomorphisms. Implementations of *num_loops* need only count the number of repeated edge labels in the list defining the *Vertex* object $\boldsymbol{v}$.

[2]Labels can be drawn from any finite set. In actual computer implementations, two obvious choices are to use the set of machine integers, or the set of *Edge* objects themselves (i.e., label each fatgraph edge with the corresponding computer representation).

1.2. Edges.

Definition 3.6. An *Edge* object $\boldsymbol{e}$ is an unordered pair of endpoints, so defined: each endpoint corresponds to a 2-tuple $(\boldsymbol{v}, a)$, where $\boldsymbol{v}$ is a vertex, and a is the index at which edge $\boldsymbol{e}$ appears within vertex $\boldsymbol{v}$ (the attachment index).

The notation *Edge*(⟨*endpoints*⟩) will be used for an *Edge* object comprising the specified endpoints.

Figure 3.2 provides a graphical illustration of the representation of fatgraph edges as *Edge* objects.

It is clear how an *Edge* object corresponds to a fatgraph edge: a fatgraph edge is made of two half-edges, each of which is uniquely identified by a pair formed by the end vertex $\boldsymbol{v}$ and the attachment index a. In the case of loops, the two ends will have the form $(\boldsymbol{v}, a)$, $(\boldsymbol{v}, a')$ where a and a' are the two distinct attachment indices at $\boldsymbol{v}$.

Example 3.7. For instance, consider the fatgraph depicted in Figure 3.2, and label the three edges with the natural numbers 0, 1, 2, starting with the bottom edge. Then the two vertices are represented by $\boldsymbol{a}$ = *Vertex*(1,0,0) and $\boldsymbol{b}$ = *Vertex*(2,1,2). Hence, the bottom edge e_0 is represented as *Edge*(($\boldsymbol{a}$,1), ($\boldsymbol{a}$,2)) because its label 0 appears at positions 1 and 2 in the *Vertex* object $\boldsymbol{a}$.

Given an *Edge* object $\boldsymbol{e}$, the *other_end*($\boldsymbol{e}$, $\boldsymbol{v}$, a) function returns the endpoint of $\boldsymbol{e}$ opposite to $(\boldsymbol{v}, a)$.

1.2.1. *Computation of the edge list.* The edge list $\boldsymbol{G}$.*edges* can be computed from the list of vertices as follows.

The total number m of edges is computed from the sum of vertex valences, and used to create a temporary array P of m lists (each one initially empty). We then incrementally turn P into a list of edge endpoints (in the form $(\boldsymbol{v}, a)$ where $\boldsymbol{v}$ is a vertex and a the attachment index) by just walking the list of vertices: $P[k]$ is the list [$(\boldsymbol{v}_k, 0)$, …, $(\boldsymbol{v}_k, z_k)$] where $\boldsymbol{v}_k$ (of valence z_k) is the k-th *Vertex* in $\boldsymbol{G}$.*vertices*. The list $\boldsymbol{G}$.*edges* is just P recast into *Edge* objects. In pseudo-code:

```
1  m ← (1/2) · Σ_{v∈G.vertices} valence(v)
2  P ← array of m empty lists
3  for v in G.vertices:
4    for (a, e) in enumerate(v):
5      append (v, a) to P[e]
6   wrap endpoints into "Edge" objects
7  G.edges ← [ Edge(p) for p in P ]
```

1.3. Boundary Cycles.

Definition 3.8. A *BoundaryCycle* object is a set of *corners* (see Figure 3.3).

A corner object $\boldsymbol{C}$ is a triple (*vertex*, *incoming*, *outgoing*), consisting of a vertex $\boldsymbol{v}$ and two indices $i = \boldsymbol{C}.incoming$, $j = \boldsymbol{C}.outgoing$ of consecutive edges (in the cyclic order at $\boldsymbol{v}$). In order to have a unique representation of any corner, we impose the condition that either $j = i + 1$, or i and j are, respectively, the ending and starting indices of $\boldsymbol{v}$ (regarded as a list).

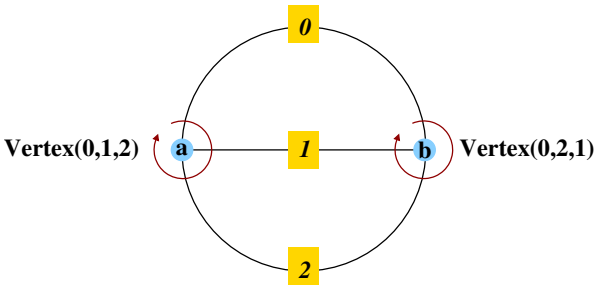


FIGURE 3.1. Representation of vertices as (cyclic) lists of edge labels; vertices are identified by lowercase Latin letters; edge labels are depicted as roman numerals on a yellow square background, sitting over the edge they label. The representation of a vertex as a list is implicitly ciliated: here we use the convention that the edge closest to the tail of the arrow is the ciliated one.

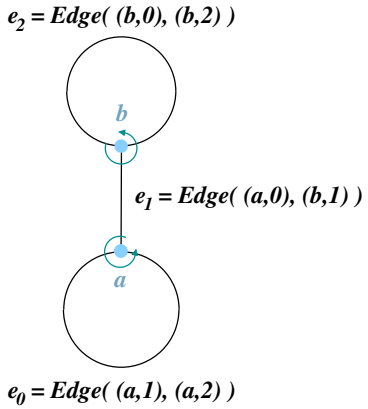


FIGURE 3.2. Representation of fatgraph edges. Each edge is identified with a pair of endpoints, where an endpoint is a vertex together with an attachment index. In the figure, letters a and b denote the vertices; attachment indices are computed by assigning index 0 to the edge closest to the orientation arrow's tail.

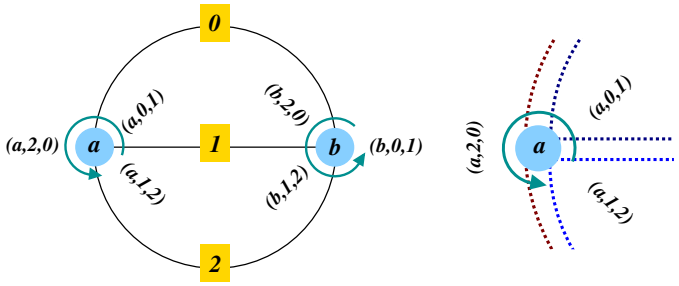


FIGURE 3.3. Representation of fatgraph boundary cycles. *Left:* How the boundary cycles are represented with corners: each boundary component is identified with the set of triplets it encloses. Therefore the boundary cycles for the graph are represented by the sets $\{(a,0,1),(b,2,0)\}$, $\{(a,1,2),(b,1,2)\}$, and $\{(a,2,0),(b,0,1)\}$. *Right:* Zoom around vertex a in the left picture, to show the three corners identified with triples (a,i,j). The indices in the triple are attachment indices, i.e., displacement relative from the ciliated edge (the one closest to the arrow tail); they bear no relation to the labels on the edges (numbers on the light yellow background in the left picture).

Example 3.9. Corner objects are meant to represent the entity formed by a vertex and two consecutive half-edges; attachment indices are used to distinguish between the two ends of a looping edge. So, for instance, the corner formed by the vertex b and edges 0 and 1 in Figure 3.3 is given by the triplet ($\boldsymbol{b}$,2,0), as edge 1 is attached at position 2 in the *Vertex* object $\boldsymbol{b}$, and edge 0 is attached at position 0; the normalization condition that $\boldsymbol{C}.\mathit{outgoing}$ be the successor of $\boldsymbol{C}.\mathit{incoming}$ in the cyclic ordering at the vertex $\boldsymbol{b}$ dictates the order of the two edges in the corner.

The notation ${}^{i}\boldsymbol{a}^{j}$ is used in Appendix C to indicate corners; a boundary cycle is denoted by $({}^{i_0}\boldsymbol{a}_0^{j_0} \to {}^{i_1}\boldsymbol{a}_1^{j_1} \to \cdots \to {}^{i_k}\boldsymbol{a}_k^{j_k})$.

It is easy to convince oneself that a *BoundaryCycle* object corresponds to a boundary cycle as defined in Section 1. Indeed, if $(L, \sigma_0, \sigma_1, \sigma_2)$ is a fatgraph, then the boundary cycles are defined as the orbits of σ_2 on the set L of half-edges; a *(endpoint vertex, attachment index)* pair uniquely identifies an half-edge and can thus be substituted for it. For computational efficiency reasons, we add an additional successor index to form the corner triple $(\boldsymbol{v}, i, j)$ so that the action of σ_2 can be computed from corner data alone, without any reference to the ambient fatgraph.[3]

Since distinct orbits are disjoint, two *BoundaryCycle* objects are either identical (they comprise the same corners) or have no intersection. In particular, this representation based on corners distinguishes boundary cycles made of the same edges: for instance, the boundary cycles of the fatgraph depicted in Figure 3.4 are represented by the disjoint set of corners $\{(\boldsymbol{v}, 2, 3), (\boldsymbol{v}, 4, 5), (\boldsymbol{v}, 0, 1)\}$ and $\{(\boldsymbol{v}, 1, 2), (\boldsymbol{v}, 3, 4), (\boldsymbol{v}, 5, 0)\}$.

1.3.1. *Computation of boundary cycles.* The procedure for computing the set of boundary cycles of a given *Fatgraph* object $\boldsymbol{G}$ is listed in Algorithm 1. The algorithm closely follows a geometrical procedure: starting with any corner, follow its "outgoing" edge to its other endpoint, and repeat until we come back to the starting corner. The list of corners so gathered is a boundary cycle. At each iteration, the used corners are cleared out of the *corners* list by replacing them with the special value USED, so that they will not be picked up again in subsequent iterations.

Lemma 3.10. *For any Fatgraph object $\boldsymbol{G}$ representing a fatgraph G, the function compute_boundary_cycles in Algorithm 1 has the following properties: 1) terminates in finite time, and 2) returns a list of BoundaryCycle objects that represent the boundary cycles of G.*

PROOF. The algorithm works on a temporary array *corners*: as it walks along a boundary cycle (lines 24–30), corner triples are moved from the working array to the *triples* list and replaced with the constant USED; when we're back to the starting corner, a *BoundaryCycle* object $\boldsymbol{b}$ is constructed from the *triples* list and appended to the result.

The *corners* variable is a list, the n-th item of which is (again) a list holding the corners around the n-th vertex (i.e., $\boldsymbol{G}.\mathit{vertices}[n]$), in the order they are encountered when winding around the vertex. By construction, $\mathit{corners}[\boldsymbol{v}][i]$ has the the form $(\boldsymbol{v}, i, j)$ where j is the index following i in the cyclic order, i.e., $(\boldsymbol{v}, i, j)$ represents the corner formed by the "incoming" i-th edge and the "outgoing" j-th edge.

[3]This is important in order to share the same corner objects across multiple *BoundaryCycle* instances, which saves computer memory.

Algorithm 1 Output the set of boundary cycles of a *Fatgraph* object $\boldsymbol{G}$. Input to the algorithm is a *Fatgraph* object $\boldsymbol{G}$; the output is a list of *BoundaryCycle* objects. The special constant USED marks locations in the temporary array *corners* whose contents has already been assigned to a boundary cycle.

```
1   def compute_boundary_cycles(G):
2       build working array of corners
3       corners ← [ [ (v, i, ((i + 1) % |v|)) for i in 0, . . . , |v| − 1 ]
4                   for v in G.vertices ]
5       result ← empty list
6       l_0 ← 0
7       i_0 ← 0
8       while True:
9           locate the first unused corner
10          for l in l_0, . . ., size(corners)−1:
11              v ← G.vertices[l]
12              i ← first_index_not_used(corners[v], i_0)
13              if i is not None:
14                  exit "for" loop
15          if l = size(corners)−1 and i is None:
16              all corners used, mission accomplished
17              return result
18          else:
19              l_0 ← l
20              i_0 ← i
21          walk the boundary cycle and record corners
22          start ← (v, i)
23          triples ← empty list
24          while (v, i) ≠ start or size(triples) = 0:
25              triples.append(corners[v][i])
26              j ← corners[v][i][2]
27              e ← v[j]
28              mark location as ''used''
29              corners[v][i] ← USED
30              (v, i) ← other_end(e, v, j)
31              b ← BoundaryCycle(triples)
32          result.append(b)
33
34  def first_index_not_used(L):
35      for index, item in enumerate(L):
36          if item is not USED:
37              return index
38      return None
```

The starting corner for each walk along a boundary cycle is determined by scanning the *corners* list (lines 10–14): loop over all indexes $\boldsymbol{v}$, i in the *corners* list, and quit looping as soon as *corners*[$\boldsymbol{v}$][i] is not USED (line 13). If all locations in the corners list are USED, then the all corners have been assigned to a boundary cycle and we can return the result list to the caller. □

Example 3.11. Consider the fatgraph in Figure 3.3: it has two vertices $\boldsymbol{a}$ = *Vertex*(0,1,2) and $\boldsymbol{b}$ = *Vertex*(0,2,1) with three incident edges each, so initially we have *corners*[$\boldsymbol{a}$] = [($\boldsymbol{a}$,0,1), ($\boldsymbol{a}$,1,2), ($\boldsymbol{a}$,2,0)] and *corners*[$\boldsymbol{b}$] = [($\boldsymbol{b}$,0,1), ($\boldsymbol{b}$,1,2), ($\boldsymbol{b}$,2,0)]. Start with the first available corner ($\boldsymbol{a}$,0,1) and follow the edge attached at position 1 to its other end ($\boldsymbol{b}$,2); the corresponding corner *corners*[$\boldsymbol{b}$][2] is ($\boldsymbol{b}$,2,0). Hence we continue the walk with edge 0 (attached at position 0 to vertex $\boldsymbol{b}$), and through its other end ($\boldsymbol{a}$,0) we come back to the starting point. So, one boundary cycle is formed by corners { ($\boldsymbol{a}$,0,1), ($\boldsymbol{b}$,2,0) }; the *corners* list now has the values *corners*[$\boldsymbol{a}$] = [USED, ($\boldsymbol{a}$,1,2), ($\boldsymbol{a}$,2,0)] and *corners*[$\boldsymbol{b}$] = [($\boldsymbol{b}$,0,1), ($\boldsymbol{a}$,1,2), USED].

1.4. Orientation.

According to Definition 2.21, orientation is given by a total order of the edges (which directly translates into an orientation of the associated orbifold cell).

Definition 3.12. The orientation $\boldsymbol{G}$.*orient* is a list that associates each edge with its position according to the order given by the orientation. Two such lists are equivalent if they differ by an even permutation.

If $\boldsymbol{e}_1$ and $\boldsymbol{e}_2$ are edges in a *Fatgraph* object $\boldsymbol{G}$, then $\boldsymbol{e}_1$ precedes $\boldsymbol{e}_2$ if and only if $\boldsymbol{G}.orient[\boldsymbol{e}_1] < \boldsymbol{G}.orient[\boldsymbol{e}_2]$; this links the fatgraph orientation from Definition 2.21 with the one above.

If a *Fatgraph* object is derived from another *Fatgraph* instance (e.g., when an edge is contracted), the resulting graph must derive its orientation from the "parent" graph, if we want the edge contraction to correspond to taking cell boundary in the orbicomplex $\mathcal{M}_{g,n}^{\mathrm{comb}}$.

When no orientation is given, the trivial one is (arbitrarily) chosen: edges are ordered in the way they are listed in the $\boldsymbol{G}$.*edges* list, i.e., $\boldsymbol{G}.orient[\boldsymbol{e}]$ is the position at which $\boldsymbol{e}$ appears in $\boldsymbol{G}$.*edges*.

According to Definition 2.22, a fatgraph is orientable iff it has no orientation-reversing automorphism. The author knows of no practical way to ascertain if a fatgraph is orientable other than enumerating all automorphisms and checking if any one of them reverses orientation:

```
1  def is_oriented(G):
2    for a in automorphisms(G):
3      if is_orientation_reversing(a):
4        return False
5    no orientation−reversing automorphism found, G is orientable
6    return True
```

1.5. A category of *Fatgraph* objects.

1.5.1. *Isomorphisms of Fatgraph objects.* In this section, we shall only give the definition of *Fatgraph* isomorphisms and prove the basic properties; the algorithmic generation and treatment of *Fatgraph* isomorphisms is postponed to Section 2.

Definition 3.13. An isomorphism of *Fatgraph* objects $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ is a triple $\boldsymbol{f} = (pv, rot, pe)$ where:

- pv is a permutation of the vertices: vertex $\boldsymbol{v}_1$ of $\boldsymbol{G}_1$ is sent to vertex $pv[\boldsymbol{v}]$ of $\boldsymbol{G}_2$, and rotated by $rot[\boldsymbol{v}]$ places leftwards;
- pe is a permutation of the edge labels: edge $\boldsymbol{e}$ in $\boldsymbol{G}_1$ is mapped to edge $pe[\boldsymbol{e}]$ in $\boldsymbol{G}_2$.

The adjacency relation must be preserved by isomorphism triples: if $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are endpoint vertices of the edge $\boldsymbol{e}$, then $pv[\boldsymbol{v}_1]$ and $pv[\boldsymbol{v}_2]$ must be the endpoint vertices of edge $pe[\boldsymbol{e}]$ in $\boldsymbol{G}_2$.

Since a vertex in a *Fatgraph* instance is essentially the list of labels of edges attached to that vertex, we can dually state this compatibility condition as requiring that, for any vertex $\boldsymbol{v}$ in $\boldsymbol{G}_1.vertices$ and any valid index j of an edge of $\boldsymbol{v}$, we have:

$$\boldsymbol{G}_2.vertices[pv[\boldsymbol{v}]][j+rot[\boldsymbol{v}]] = pe[\boldsymbol{G}_1.vertices[\boldsymbol{v}][j]] \tag{1.1}$$

The above formula (1.1) makes the parallel between *Fatgraph* object isomorphisms and fatgraph maps (in the sense of Definition 2.1) explicit.

Lemma 3.14. *Let G_1, G_2 be fatgraphs, represented respectively by $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$. Every isomorphism of fatgraphs $f\colon G_1 \to G_2$ lifts to a corresponding isomorphism $\boldsymbol{f} = (pv, rot, pe)$ on the computer representations. Conversely, every triple (pv, rot, pe) representing an isomorphism between the Fatgraph instances induces a (possibly trivial) fatgraph isomorphism between G_1 and G_2.*

Proof. Every isomorphism $f\colon G_1 \to G_2$ naturally induces bijective maps $f_V\colon V(G_1) \to V(G_2)$ and $f_E\colon E(G_1) \to E(G_2)$ on vertices and edges. Given a cilium on every vertex, f additionally determines, for each vertex $v \in V(G)$, the displacement $f_{rot}(v)$ of the image of the cilium of v relative to the cilium of $f_V(v)$. Similarly, f_E determines a bijective mapping of edge labels, and is completely determined by it. This is exactly the data collected in the triple $(pv, rots, pe)$, and the compatibility condition (1.1) holds by construction.

Conversely, assume we are given a triple $(pv, rots, pe)$, representing an isomorphism of *Fatgraph* instances. We can construct maps f_V, f_E as follows: f_V sends a vertex $\boldsymbol{v} \in \boldsymbol{G}_1$ to the vertex corresponding to $pv[\boldsymbol{v}]$; f_E maps the cilium of $\boldsymbol{v}$ to the edge attached to $pv[\boldsymbol{v}]$ at $rot[\boldsymbol{v}]$ positions away from the cilium; the compatibility condition (1.1) guarantees that f_E is globally well-defined. □

Example 3.15. Consider the fatgraph G in Figure 3.1 again; a *Fatgraph* object $\boldsymbol{G}$ representing it has vertices $\boldsymbol{a}$= *Vertex*(0,1,2), $\boldsymbol{b}$= *Vertex*(0,2,1), and edges $\boldsymbol{e}_0$, $\boldsymbol{e}_1$, $\boldsymbol{e}_2$. Picture G as embedded on the surface of a sphere, with the vertices on the two poles; an obvious automorphism is the one induced by rotation of the sphere along its axis: f_E maps $\boldsymbol{e}_0 \mapsto \boldsymbol{e}_1$, $\boldsymbol{e}_1 \mapsto \boldsymbol{e}_2$, and $\boldsymbol{e}_2 \mapsto \boldsymbol{e}_0$, hence $pe[\boldsymbol{e}_0] = \boldsymbol{e}_1$, etc. Correspondingly, we have that f_V fixes the vertices but rotates the attachment indices: $pv[\boldsymbol{a}] = \boldsymbol{a}$ and $pv[\boldsymbol{b}] = \boldsymbol{b}$, but $rot[\boldsymbol{a}] = 1$ and $rot[\boldsymbol{b}] = 1$.

Exchanging the north and south poles of the sphere yields a different automorphism f' of the graph G: this time we have clearly $\boldsymbol{f}'.pv[\boldsymbol{a}] = \boldsymbol{b}$ and $pv[\boldsymbol{b}] = \boldsymbol{a}$ with $\boldsymbol{f}'.rot[\boldsymbol{a}] = 0$ and $rot[\boldsymbol{b}] = 0$, but $\boldsymbol{f}'.pe[\boldsymbol{e}_1] = \boldsymbol{e}_2$, $pe[\boldsymbol{e}_2] = \boldsymbol{e}_1$, and $pe[\boldsymbol{e}_0] = \boldsymbol{e}_0$ by (1.1).

Lemma 3.16. *Let $\boldsymbol{G}_1$, $\boldsymbol{G}_2$ be Fatgraph objects, and η a bijective map between $\boldsymbol{G}_1$.edges and $\boldsymbol{G}_2$.edges that preserves the incidence relation. Then there is a unique Fatgraph isomorphism $\boldsymbol{f}$ that extends η (in the sense that $\boldsymbol{f}.\mathrm{pe} = \eta$).*

PROOF. Start constructing the *Fatgraph* morphism $\boldsymbol{f}$ by setting $\boldsymbol{f}.pe = \eta$. If $\boldsymbol{e}_1, \ldots, \boldsymbol{e}_{z_k}$ are the edges incident to $\boldsymbol{v}_k \in \boldsymbol{G}_1.vertices$, then there is generally one and only one endpoint $\boldsymbol{v}'_k$ common to edges $\eta(\boldsymbol{e}_k)$; define $\boldsymbol{f}.pv[\boldsymbol{v}_k] = \boldsymbol{v}'_k$. There is only one case in which this is not true, namely, if *all* edges share the same two endpoints:[4] in this case, however, there is still only one choice of $\boldsymbol{f}.pv[\boldsymbol{v}_k]$ such that the cyclic order of edges at the source vertex matches the cyclic order of edges at the target vertex. Finally, choose $\boldsymbol{f}.rot[\boldsymbol{v}_k]$ as the displacement between the cilium at $\boldsymbol{v}'_k$ and the image of the cilium of $\boldsymbol{v}_k$.

It is easy to check that eq. (1.1) holds, so $\boldsymbol{f}$ is a well-defined isomorphism. □

1.5.2. *Contraction morphisms.* Recall from the definition in Section 1.2 that contraction produces a "child" fatgraph from a "parent" fatgraph and a chosen regular (i.e., non-looping) edge.

The *Fatgraph.contract* method (see Algorithm 2) thus takes as input the "parent" graph $\boldsymbol{G}$ and the edge $\boldsymbol{e}$ to contract, and produces as output the "child" fatgraph $\boldsymbol{G}'$. The contraction algorithm proceeds in the following way:

- The two end vertices of the edge $\boldsymbol{e}$ are fused into one: the list $\boldsymbol{G}'.vertices$ is built by copying the list $\boldsymbol{G}.vertices$, removing the two endpoints of $\boldsymbol{e}$, and adding the new vertex (resulting from the collapse of $\boldsymbol{e}$) at the end.
- Deletion of an edge also affects the orientation: the orientation $\boldsymbol{G}'.orient$ on the "child" fatgraph keeps the edges in the same order as they are in the parent fatgraph. However, since $\boldsymbol{G}'.orient$ must be a permutation of the edge indices, we need to renumber the edges and shift the higher-numbered edges down one place.
- The "child" graph $\boldsymbol{G}'$ is constructed from the list $\boldsymbol{G}'.vertices$ and the derived orientation $\boldsymbol{G}'.orient$; the list of "new" edges is constructed according to the procedure given in Section 1.2.1.

Listing 2 summarizes the algorithm applied.

The vertex resulting from the contraction of $\boldsymbol{e}$ is formed as follows. Assume $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ are the endpoint vertices of the contracted edge. Fuse the endpoints of the contracted edge as follows:

(1) Rotate the lists $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ so that the given edge $\boldsymbol{e}$ appears *last* in $\boldsymbol{v}_1$ and *first* in $\boldsymbol{v}_2$.
(2) Form the new vertex $\boldsymbol{v}$ by concatenating the two rotated lists (after expunging $\boldsymbol{e}$).

Note that this changes the attachment indices of all edges incident to $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, therefore the edge list of $\boldsymbol{G}'$ needs to be recomputed from the vertex list.

Example 3.17. Consider the fatgraph G on the left of Figure 3.5; assume we want to contract the edge $\boldsymbol{e}_1$. According to the above recipe, the first step is to use cyclic invariance to rewrite the vertices $\boldsymbol{a}$ and $\boldsymbol{b}$ in the form: $\boldsymbol{a}$= *Vertex*(2,0,1) and $\boldsymbol{b}$= *Vertex*(1,0,2); then the two vertices are fused into one: $\boldsymbol{a}'$= *Vertex*(2,0,0,2), which is the only vertex of the contracted fatgraph G'.

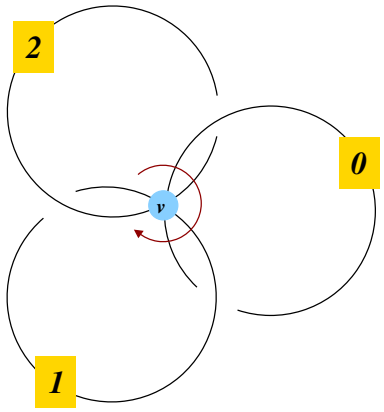


FIGURE 3.4. A fatgraph whose two boundary cycles are comprised of exactly the same edges; however, they give rise to disjoint sets of corners: $\{(\boldsymbol{v}, 2, 3), (\boldsymbol{v}, 4, 5), (\boldsymbol{v}, 0, 1)\}$ versus $\{(\boldsymbol{v}, 1, 2), (\boldsymbol{v}, 3, 4), (\boldsymbol{v}, 5, 0)\}$.

Algorithm 2 Construct a new *Fatgraph* object $\boldsymbol{G}'$ obtained by contracting the edge $\boldsymbol{e}$ in $\boldsymbol{G}$. The renumbering function s is the identity on numbers in the range $0, \ldots, e-1$, and shifts numbers in range $e+1, \ldots, m$ down by 1. Function $rotated(L, p)$ returns a copy of list L shifted leftwards by p places.

```
1  def contract(G, e):
2    let (v_1, a_1), (v_2, a_2) be the endpoints of e
3    V' ← [ Vertex(x for x in v if x ≠ e)
4             for v in G.vertices if v ≠ v_1 and v ≠ v_2 ]
5     append the fused vertex at end of list V
6    v' ← Vertex(rotated(v_1, a_1) + rotated(v_2, a_2))
7    V'.append(v')
8    ω' ← [ s(G'.orient[x]) for x in G'.edges if x ≠ e ]
9    return Fatgraph(vertices ← V'; orient ← ω')
```

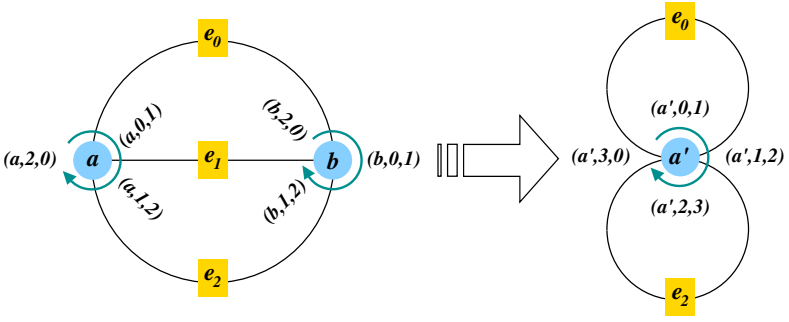


FIGURE 3.5. Example of a contraction morphism. Contract edge $\boldsymbol{e}_1$ in the "parent" fatgraph on the left to obtain the "child" fatgraph on the right. Annotations show how the corners are changed by the contraction procedure.

The "child" fatgraph G' inherits an orientation from the "parent" fatgraph, which might differ from its default orientation. Let $\alpha_1, \ldots, \alpha_h, \ldots, \alpha_m$ be the edges of the parent fatgraph G, with $e = \alpha_h$ being contracted to create the "child" graph G'. If $\alpha_{k(1)} < \alpha_{k(2)} < \ldots < \alpha_{k(m)}$ is the ordering on $E(G)$ that induces the orientation on G and $h = k(j)$, then $\alpha_{k(1)} < \ldots < \alpha_{k(j-1)} < \alpha_{k(j+1)} < \ldots < \alpha_{k(m)}$ descends to a total order on the edges of G' and induces the correct orientation.[5]

Orientation is represented in a *Fatgraph* object as a list, mapping edge labels to a position in the total order; using the notation above, the orientation of G is given by $\omega := k^{-1}$. The orientation on G' is then given by ω' defined as follows:

$$\omega'(i) := \begin{cases} \omega(i) & \text{if } \omega(i) < h, \\ \omega(i) - 1 & \text{if } \omega(i) > h. \end{cases}$$

Alternatively we can write:

$$\omega' = s \circ \omega, \qquad s(x) := \begin{cases} x & \text{if } x < h, \\ x - 1 & \text{if } x > h. \end{cases}$$

This corresponds exactly to the assignment in Algorithm 2.

Example 3.18. Building upon Example 3.17, assume the orientation on G is $\omega = (0 < 1 < 2)$, meaning that the edges $\boldsymbol{e}_i$ are ordered in such a way that $\boldsymbol{e}_i < \boldsymbol{e}_j$ iff $i < j$. Let $\boldsymbol{e}'_0$, $\boldsymbol{e}'_1$ be the edges of the "child" fatgraph G', with $\boldsymbol{e}'_0$ being the image of $\boldsymbol{e}_0$, and $\boldsymbol{e}'_1$ the image of $\boldsymbol{e}_2$. So the orientation induced on G' is $\boldsymbol{e}'_0 < \boldsymbol{e}'_1$, i.e., $\omega' = (0 < 1)$.

The above discussion can be summarized in the following.

Lemma 3.19. *If $\boldsymbol{G}$ and $\boldsymbol{G}'$ represent fatgraphs G and G', and $\boldsymbol{G} = \mathit{contract}(\boldsymbol{G}', \boldsymbol{e})$, then G is obtained from G' by contraction of the edge e represented by $\boldsymbol{e}$.*

The contract_boundary_cycle function. The boundary cycles of the "child" *Fatgraph* object $\boldsymbol{G}'$ can also be computed from those of $\boldsymbol{G}$. The implementation (see Listing 1) is quite straightforward: we copy the given list of corners and alter those who refer to the two vertices that have been merged in the process of contracting the specified edge.

Let $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ be the end vertices of the edge to be contracted, and a_1, a_2 be the corresponding attachment indices. Let z_1 and z_2 be the valences of vertices $\boldsymbol{v}_1$, $\boldsymbol{v}_2$. We build the list of corners of the boundary cycle in the "child" graph incrementally: the $\boldsymbol{b}'$ lists starts empty (line 6), and is then added corners as we run over them in the loop between lines 7 and 26.

There are four distinct corners that are bounded by the edge $\boldsymbol{e}$ to be contracted; denote them by $\boldsymbol{C}_1$, $\boldsymbol{C}_2$, $\boldsymbol{C}_3$, $\boldsymbol{C}_4$. These map onto two distinct corners $\boldsymbol{C}$, $\boldsymbol{C}'$ after contraction. Assume that $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ map to $\boldsymbol{C}$: then $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ lie "on the same side" of the contracted edge, i.e., any boundary cycle that includes $\boldsymbol{C}_1$ will include also $\boldsymbol{C}_2$ and viceversa. (See Figure 3.6 for an illustration.) Since they both map to the same corner $\boldsymbol{C}$ in the "child" graph, we only need to keep one: we choose to keep (and transform) the corner that has the contracted edge at the *second* index (lines 9–10); similarly for $\boldsymbol{C}_3$ and $\boldsymbol{C}_4$ in mapping to $\boldsymbol{C}'$ (lines 16–17).

[4]So there are only two vertices in total, and the corresponding fatgraph belongs in $\mathcal{R}_{0,m}$.

[5]That is to say, the orientation that corresponds to the orientation induced on the cell $\Delta(G')$ as a face of $\Delta(G)$.

Algorithm 3 Return a new *BoundaryCycle* instance, image of $\boldsymbol{b}$ under the topological map that contracts the edge with index $\boldsymbol{e}$.

```
1  def contract_boundary_cycle(G, b, e):
2    let (v_1, a_1), (v_2, a_2) be the endpoints of e
3    z_1 ← valence(v_1)
4    z_2 ← valence(v_2)
5     "child" boundary cycle b' starts off as an empty list
6    b' ← [ ]
7    for corner in b:
8      if corner[0] = v_1:
9        if a_1 = corner.incoming:
10         continue with next corner
11       else:
12         i_1 ← (corner.incoming − a_1 − 1) % z_1
13         i_2 ← (corner.outgoing − a_1 − 1) % z_1
14         append corner (v_1, i_1, i_2) to b'
15     elif corner[0] = v_2:
16       if a_2 = corner.incoming:
17         continue with next corner
18       if a_2 = corner.outgoing:
19         append (v_1, z_1 + z_2 − 3, 0) to b'
20       else:
21         i_1 ← z_1 − 1 + ((corner.incoming − a_2 − 1) % z_2)
22         i_2 ← z_1 − 1 + ((corner.outgoing − a_2 − 1) % z_2)
23         append (v_1, i_1, i_2) to b'
24     else:
25        keep corner unchanged
26       append corner to b'
27   return BoundaryCycle(b')
```

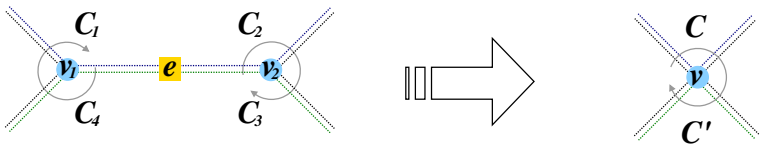


FIGURE 3.6. How corners are modified by edge contraction. *Left:* Four distinct corners are formed at the endpoints $\boldsymbol{v}_1$, $\boldsymbol{v}_2$ of edge $\boldsymbol{e}$, which is to be contracted: $\boldsymbol{C}_1 = (\boldsymbol{v}_1, 0, 2)$, $\boldsymbol{C}_2 = (\boldsymbol{v}_2, 0, 1)$, $\boldsymbol{C}_3 = (\boldsymbol{v}_2, 1, 2)$, and $\boldsymbol{C}_4 = (\boldsymbol{v}_1, 0, 1)$. Edges are shown thickened, and (potentially) distinct boundary cycles are drawn in different colors. *Right:* After contraction of $\boldsymbol{e}$, corners $\boldsymbol{C}_1$ and $\boldsymbol{C}_2$ are fused into $C = (\boldsymbol{v}, 0, 1)$, and $\boldsymbol{C}_3$, $\boldsymbol{C}_4$ are fused into $\boldsymbol{C}' = (\boldsymbol{v}, 2, 3)$.

Recall that, when contracting an edge with endpoints $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, the new vertex is formed by concatenating two series of edges: *(1)* edges attached to the former $\boldsymbol{v}_1$, starting with the successor (in the cyclic order) of the contracted edge; *(2)* edges attached to the former $\boldsymbol{v}_2$, starting with the successor of the contracted edge. Therefore:

(1) The image of a corner rooted in vertex $\boldsymbol{v}_1$ will have its attachment indices rotated leftwards by $a_1 + 1$ positions: the successor of the contracted edge has now attachment index 0 (lines 12–13). Note that the highest attachment index belonging into this group is $z_1 - 2$: position $z_1 - 1$ would correspond to the contracted edge.
(2) The image of a corner rooted in vertex $\boldsymbol{v}_2$ has its attachment indices rotated leftwards by $a_2 + 1$ positions, and shifted up by $z_1 - 1$ (lines 21–22). As a special case, when the contracted edge is in second position we need to map the corner to the corner having attachment index 0 in second position (line 19).

Any other corner is copied with no alterations (line 26).

Worked out examples are provided in Figures 3.5 and 3.6.

1.5.3. *The category of Fatgraph objects.* We can now formally define a category of *Fatgraph* objects and their morphisms.

Definition 3.20. $\mathcal{R}^{\#}$ is the category whose objects are *Fatgraph* objects, and whose morphisms are compositions of *Fatgraph* isomorphisms (as defined in Section 2) and edge contraction maps.

More precisely, if $\boldsymbol{G}$ and $\boldsymbol{G}'$ are isomorphic *Fatgraph* objects, then the morphism set $\mathcal{R}^{\#}(\boldsymbol{G}, \boldsymbol{G}')$ is defined as the set of *Fatgraph* isomorphisms in the sense of Section 2; otherwise, let m and m' be the number of edges of $\boldsymbol{G}$, $\boldsymbol{G}'$, and set $k := m - m'$: each element in $\mathcal{R}^{\#}(\boldsymbol{G}, \boldsymbol{G}')$ has the form $\boldsymbol{a}' \circ (\pi_1 \circ \cdots \circ \pi_k) \circ \boldsymbol{a}$ where $\boldsymbol{a}$, $\boldsymbol{a}'$ are automorphisms of $\boldsymbol{G}$, $\boldsymbol{G}'$ (respectively) and $\pi_1, \ldots, \pi_k$ are non-loop edge contractions.

Theorem 3.21. *There exists a functor K from the category $\mathcal{R}^{\#}$ of Fatgraph objects to the category $\mathcal{R}$ of abstract fatgraphs, which is surjective and full.*

Proof. Given a *Fatgraph* $\boldsymbol{G}$, its constituent *Vertex* objects determine cyclic sequences $v_0 = (e^0_0, e^0_1, \ldots, e^0_{z_0}), \ldots, v_l = (e^l_0, \ldots, e^l_{z_l})$, such that

$$\{e^0_0, \ldots, e^0_{z_0}, e^1_0, \ldots, e^{l-1}_{z_{l-1}}, e^l_0, \ldots, e^l_{z_l}\} = \{0, \ldots, m-1\}.$$

Fix a starting element for each of the cyclic sequences $v_0, \ldots, v_l$. Then set:

$$L := \{(e, i, v) : v = v_j \in \{v_0, \ldots, v_l\},\ e = e^j_i \in v\},$$

and define maps $\sigma_0, \sigma_1, \sigma_2 \colon L \to L$ as follows:

» σ_0 sends (e, i, v_j) to (e', i', v_j) where $i' = (i+1)\%z_j$ and $e' = e^j_{i'}$ is the successor of e in the cyclic order at v_j;
» σ_1 maps (e, i, v) to the unique other triplet $(e', i', v') \in L$ such that $e = e'$;
» finally, σ_2 is determined by the constraint $\sigma_0 \circ \sigma_2 = \sigma_1$.

Then $K(\boldsymbol{G}) = (L, \sigma_0, \sigma_1, \sigma_2)$ is a fatgraph. Figure 3.7 provides a graphical illustration of the way a *Fatgraph* object is constructed out of such combinatorial data.

Now let G be an abstract fatgraph; assuming G has m edges, assign to each edge a "label", i.e., pick a bijective map $e : E(G) \to E$, where E is an arbitrary finite set. Each vertex $v \in V(G)$ is thus decorated with a cyclic sequence of edge labels; the set of which determines a *Fatgraph* object $\boldsymbol{G}$; it is clear that $G = K(\boldsymbol{G})$.

This proves that K is surjective; since every fatgraph morphism can be written as a composition of isomorphisms and edge contractions (Lemma 2.4), it is also full. It is clear that every edge contraction is the image of an edge contraction in the corresponding *Fatgraph* objects, and the assertion for isomorphisms follows as a corollary of Lemma 3.14. □

Definition 3.22. If $G = K(\boldsymbol{G})$ then we say that the *Fatgraph* object $\boldsymbol{G}$ *represents* the abstract fatgraph G.

It is clear from the construction above that there is a considerable amount of arbitrary choices to be made in constructing a representative *Fatgraph*; there are thus many representatives for the same fatgraph, and different choices lead to equivalent *Fatgraph* objects.

Lemma 3.23. *Two distinct Fatgraph objects representing the same abstract fatgraph are isomorphic.*

Proof. Assume $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ both represent the same abstract fatgraph $G = K(\boldsymbol{G}_1) = K(\boldsymbol{G}_2)$. Let η_1, η_2 be the maps that send *Edge* objects in $\boldsymbol{G}_1$, $\boldsymbol{G}_2$ to the corresponding edges in G; then $\eta = \eta_1^{-1} \circ \eta_2$ maps edges of $\boldsymbol{G}_1$ into edges of $\boldsymbol{G}_2$ and respects the incidence relation, therefore it is the edge part of a *Fatgraph* isomorphism by Lemma 3.16. □

Theorem 3.24. *The categories $\mathcal{R}^{\#}$ and $\mathcal{R}$ are equivalent.*

Proof. The functor K is surjective and full by Theorem 3.21; that it is also faithful follows from the following argument. Any fatgraph morphism is a composition of edge contractions and isomorphisms. Any isomorphism determines, in particular, a map on the set of edges, and there is one and only one *Fatgraph* isomorphism induced by this map (Lemma 3.16). Any edge contraction is uniquely determined by the contracted edge: if $f\colon G_1 \to G_2$ is the morphism contracting edge e and $G_i = K(\boldsymbol{G}_i)$, then $\boldsymbol{f}$, contraction of the *Edge* object $\boldsymbol{e}$ representing e, is the sole morphism of $\boldsymbol{G}_1$ into $\boldsymbol{G}_2$ that maps onto f. □

2. Fatgraphs isomorphism and equality testing

The isomorphism problem on computer representations of fatgraphs consists in finding out when two distinct *Fatgraph* instances represent isomorphic fatgraphs (in the sense of Definition 2.1) or possibly the same fatgraph. Indeed, the procedure for associating a *Fatgraph* instance to an abstract fatgraph (see Theorem 3.21) involves labeling all edges, choosing a starting edge (cilium) on each vertex, and enumerating all vertices in a certain order; for each choice we get a different *Fatgraph* instance representing the same (abstract) fatgraph.

The general isomorphism problem for (ordinary) graphs is a well-known difficult problem. However, the situation is much simpler for fatgraphs, because of the following property.

Lemma 3.25 (Rigidity Property)**.** *Let G_1, G_2 be connected fatgraphs, and $f\colon G_1 \to G_2$ an isomorphism. For any vertex $v \in V(G_1)$, and any edge x incident to v, f is uniquely determined (up to homotopies fixing the vertices of G_i) by its restriction to v and x.*

In particular, an isomorphism of graphs with ciliated vertices is completely determined once the image $w = f(v)$ of a vertex v is known, together with the displacement (relative to the cyclic order at w) of the image of the cilium of v relative to the cilium of the image vertex w.

PROOF. Consider f as a CW-complex morphism: $f = (f_0, f_1)$ where f_i is a continuous map on the set of i-dimensional cells.

Let U be a small open neighborhood of $v \in V(G_1)$. Given $f|_U$, incrementally construct a CW-morphism $f'\colon G_1 \to G_2$ as follows. Each edge x' incident to v can be expressed as $x' = \sigma_0^\alpha x$ for some $0 \leqslant \alpha <$*valence*(v). Let $w = f(v)$ and $y = f(x)$, and define:

$$\begin{aligned} f_1'(x) &:= y, \\ f_0'(v) &:= w, \\ f_1'(x') &:= \sigma_0^\alpha y = \sigma_0^\alpha f_1'(x) \quad \text{if } x' = \sigma_0^\alpha x, \\ f_0'(v_\alpha') &:= w_\alpha', \end{aligned}$$

where:

» v_α' is the endpoint of $x' = \sigma_0^\alpha x$ "opposite" to v,
» w_α' is the endpoint of $y' = \sigma_0^\alpha y$ "opposite" to w.

Then f' extends f on a closed set $U' \supsetneq U$, which contains the subgraph formed by all edges attached to v and v', and their endpoints. In addition:

» $f_1'(x') = f(x')$ up to a homotopy fixing the endpoints since f commutes with σ_0,
» $f_0'(v_\alpha) = f(v_\alpha)$ since f preserves adjacency.

By repeating the same construction about the vertices v_α' and w_α', one can extend f' to a CW-morphism that agrees with f on an closed set U'' such that U' is strictly contained in the interior of U''.

Recursively, by connectedness, we can thus extend f' to agree with f (up to homotopy) over all of G_1. □

2.1. Enumeration of *Fatgraph* isomorphisms. The stage is now set for presenting the algorithm to enumerate the isomorphisms between any two given *Fatgraph* objects. Pseudo-code is listed in Algorithm 4; as this procedure is quite complex, a number of auxiliary functions have been used, whose purpose is explained in Section 2.1.1. Function *isomorphisms*, given two *Fatgraph* objects $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$, returns a list of triples (*pv*, *rot*, *pe*), each of which determines an isomorphism. If there is no isomorphism connecting the two graphs, then the empty list [ ] is returned.

By the rigidity Lemma 3.25, any fatgraph isomorphism is uniquely determined by the mapping of a small neighborhood of any vertex. The overall strategy of the

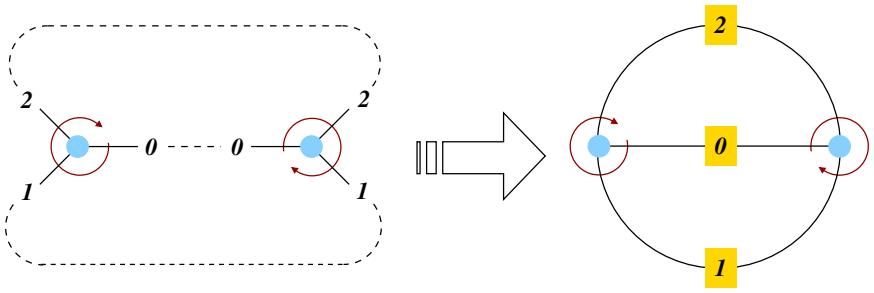


FIGURE 3.7. Construction of a fatgraph out of a set of *Vertex* instances: half-edges tagged with the same (numeric) label are joined together to form an edge.

Algorithm 4 Enumerate isomorphisms between two *Fatgraph* objects $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$: output of the algorithm is a list of triples *(pv, rot, pe)*. If there is no isomorphism connecting the two input fatgraphs, the empty list is returned.

```
1  def isomorphisms(G_1, G_2):
2      immediately rule out easy cases of no isomorphisms
3    if graphs invariants differ:
4      return [ ]
5    result ← [ ]
6    vs1 ← valence_spectrum(G_1)
7    vs2 ← valence_spectrum(G_2)
8    (valence, vertices) ← starting_vertices(G_2)
9    v_1 ← vs1[valence][0]
10   for v_2 in compatible_vertices(v_1, vertices):
11     for rot in 0, . . . ,valence:
12       Initialize pv, rots, pe as empty maps
13       pv[v_1] ← v_2
14       rots[v_1] ← rot
15       extend_map(pe, v_1, rotated(v_2, rot))
16       if extension failed:
17         continue with next rot
18        breadth−first search to extend the mapping over corresponding vertices
19       nexts ← neighbors(pv, pe, G_1, v_1, G_2, v_2)
20       while size(pv) < G_1.num_vertices:
21         neighborhood ← [ ]
22         for (v'_1, v'_2, r) in nexts:
23           (pv, rots, pe) ← extend_iso(pv, rots, pe, G_1, v'_1, r, G_2, v'_2)
24           if cannot extend:
25             exit "while" loop and continue with next rot
26           append neighbors(pv, pe, G_1, v'_1, G_2, v'_2) to neighborhood
27         nexts ← neighborhood
28       isomorphism found, record it
29       result.append((pv, rots, pe))
30   return result
```

algorithm is thus to pick a pair of "compatible" vertices and try to extend the map as in the proof of Lemma 3.25.

We wish to stress the difference with isomorphism of ordinary graphs: since an isomorphism f is uniquely determined by any pair of corresponding vertices, the initial choice of candidates v, $f(v)$ either yields an isomorphism or it does not: there is no backtracking involved.

Since the isomorphism computation is implemented as an exhaustive search, it is worth doing a few simple checks to rule out cases of non-isomorphic graphs (lines 3–4). One has to weigh the time taken to compute a graph invariant versus the potential speedup obtained by not running the full scan of the search space; experiments run using the Python code show that the following simple invariants already provide some good speedup:

» the number of vertices, edges, boundary cycles;
» the total number of loops;
» the set of valences;
» the number of vertices of every given valence.

Since an isomorphism is uniquely determined by its restriction to *any* vertex, one can restrict to considering just pairs of the form $(\boldsymbol{v}_1, \boldsymbol{v}_2)$ where $\boldsymbol{v}_1$ is a chosen vertex in $\boldsymbol{G}_1$. Then the algorithm tries all possible ways (rotations) of mapping $\boldsymbol{v}_1$ into a compatible vertex $\boldsymbol{v}_2$ in $\boldsymbol{G}_2$. The body of the inner loop (line 11 onwards) mimics the construction in the proof of Lemma 3.25.

The starting vertex $\boldsymbol{v}_1$ should be selected so to minimize the number of mapping attempts performed; this is currently done by minimizing the product of valence and number of vertices of that valence on $\boldsymbol{G}_2$ (line 8), and then picking a vertex of the chosen valence in $\boldsymbol{G}_1$ as $\boldsymbol{v}_1$ (line 9).[6]

First, given the target vertex $\boldsymbol{v}_2$ and a rotation *rot*, a new triple (*pv*,*rots*,*pe*) is created; *pv* is set to represent the initial mapping of $\boldsymbol{v}_1$ onto $\boldsymbol{v}_2$, rotated leftwards by *rot* positions, and *pe* maps edges of $\boldsymbol{v}_1$ into corresponding edges of the rotated $\boldsymbol{v}_2$. If this mapping is not possible (e.g., $\boldsymbol{v}_1$ has a loop and $\boldsymbol{v}_2$ does not, or not in a corresponding position), then the attempt is aborted and execution continues from line 11 with the next candidate *rot*.

The mapping defined by (*pv*,*rots*,*pe*) is then extended to neighbors of the vertices already inserted. This entails a breadth-first search[7] over pairs of corresponding vertices, starting from $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$. Note that, in this extension step, not only the source and target vertices, but also the rotation to be applied is uniquely determined: chosen a vertex $\boldsymbol{v}_1'$ connected to $\boldsymbol{v}_1$ by an edge $\boldsymbol{e}$, there is a unique rotation r on $\boldsymbol{v}_2'$ such that $pv[\boldsymbol{e}]$ has the same attachment index to $\boldsymbol{v}_2'$ that $\boldsymbol{e}$ has to $\boldsymbol{v}_1$. If, at any stage, the extension of the current triple (*pv*, *rots*, *pe*) fails, the triplet is discarded and execution continues from line 11 with the next value of *rot*.

When the loop started at line 10 is over, execution reaches the end of the *isomorphisms* function, and returns the (possibly empty) list of isomorphisms to the caller.

[6]The checks already performed ensure that $\boldsymbol{G}_1$ and $\boldsymbol{G}_2$ have the same "valence spectrum", so $\boldsymbol{G}_1$ has at least one vertex of the chosen valence.

[7]The variables *nexts* and *neighborhood* play the role of the FIFO in the usual formulation of breadth-first search: vertices are added to *neighborhood* during a loop, and the resulting list is then orderly browsed (as *nexts*) in the next iteration.

Algorithm 5 Enumerate the candidate extensions of the given pv and pe in the neighborhood of input vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$.

```
1   def neighbors(pv, pe, G_1, v_1, G_2, v_2):
2     result ← [ ]
3     for each non-loop edge e attached to v_1:
4       let (v'_1, a_1) be the endpoint of e distinct from v_1
5       if v'_1 already in pv domain:
6         continue with next e
7       let (v'_2, a_2) be the endpoint of e'=pe[e] distinct from v_2
8       if v'_2 already in pv image:
9         continue with next e
10      result.append((v'_1, v'_2, a_1 − a_2))
11    return result
```

Theorem 3.26. *Given Fatgraph objects $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, function isomorphisms returns all Fatgraph isomorphisms from $\boldsymbol{G}_1$ to $\boldsymbol{G}_2$.*

PROOF. Given an isomorphism $f : \boldsymbol{G}_1 \to \boldsymbol{G}_2$, restrict f to the starting vertex $\boldsymbol{v}_1$: then f will be output when Algorithm 4 examines the pair $\boldsymbol{v}_1$, $f(\boldsymbol{v}_1)$; since Algorithm 4 performs an exhaustive search, f will not be missed.

Conversely, since equation (1.1) holds by construction for all the mappings returned by *isomorphisms*, then each returned triple $\boldsymbol{f}$ =(pv, *rots*, pe) is an isomorphism. □

2.1.1. *Auxiliary functions.* Here is a brief description of the auxiliary functions used in the listing of Algorithm 4 and 5. Apart from the *neighbors* function, they are all straightforward to implement, so only a short specification of the behavior is given, with no accompanying pseudo-code.

The neighbors function. A preliminary definition is necessary.

Definition 3.27. Define a candidate extension as a triplet ($\boldsymbol{v}'_1$, $\boldsymbol{v}'_2$, r), where:

- $\boldsymbol{v}'_1$ is a vertex in $\boldsymbol{G}_1$, connected to $\boldsymbol{v}_1$ by an edge $\boldsymbol{e}$;
- $\boldsymbol{v}'_2$ is a vertex in $\boldsymbol{G}_2$, connected to $\boldsymbol{v}_2$ by edge $\boldsymbol{e}' = pe[\boldsymbol{e}]$;
- r is the rotation to be applied to $\boldsymbol{v}'_2$ so that edge $\boldsymbol{e}$ and $\boldsymbol{e}'$ have the same attachment index, i.e., they are incident at corresponding positions in $\boldsymbol{v}'_1$ and $\boldsymbol{v}'_2$.

Function *neighbors* lists candidate extensions that extend map pv in the neighborhood of given input vertices $\boldsymbol{v}_1$ (in the domain fatgraph $\boldsymbol{G}_1$) and $\boldsymbol{v}_2$ (in the image fatgraph $\boldsymbol{G}_2$). It outputs a list of triplets $(\boldsymbol{v}'_1, \boldsymbol{v}'_2, r)$, each representing a candidate extension.

A sketch of this routine is given in Algorithm 5. Two points are worth of notice:

(1) By the time *neighbors* is called (at lines 19 and 26 in Algorithm 4), the map pe has already been extended over all edges incident to $\boldsymbol{v}_1$, so we can safely set $\boldsymbol{e}' = pe[\boldsymbol{e}]$ in *neighbors*.
(2) Algorithm 4 only uses *neighbors* with the purpose of extending pv and pe, so *neighbors* ignores vertices that are already in the domain or image of pv.

The valence_spectrum function. The auxiliary function *valence_spectrum*, given a *Fatgraph* instance $\boldsymbol{G}$, returns a mapping that associates to each valence z the list V_z of vertices of $\boldsymbol{G}$ with valence z.

The starting_vertices function. For each pair (z, V_z) in the valence spectrum, define its *intensity* as the product $z \cdot |V_z|$ (valence times the number of vertices with that valence). The function *starting_vertices* takes as input a *Fatgraph* object $\boldsymbol{G}$ and returns the pair (z, V_z) from the valence spectrum that minimizes intensity. In case of ties, the pair with the largest z is chosen.

The compatible and compatible_vertices functions. Function *compatible* takes a pair of vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ as input, and returns boolean *True* iff $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$ have the same invariants. (This is used as a short-cut test to abandon a candidate mapping before trying a full adjacency list extension, which is computationally more expensive.) The sample code uses valence and number of loops as invariants.

The function *compatible_vertices* takes a vertex $\boldsymbol{v}$ and a list of vertices L, and returns the list of vertices in L that are compatible with $\boldsymbol{v}$ (i.e., those which $\boldsymbol{v}$ could be mapped to).

The extend_map and extend_iso functions. The *extend_map* function takes as input a mapping *pe* and a pair of *ciliated* vertices $\boldsymbol{v}_1$ and $\boldsymbol{v}_2$, and alters *pe* to map edges of $\boldsymbol{v}_1$ to corresponding edges of $\boldsymbol{v}_2$: the cilium to the cilium, and so on: $pe[\sigma_0^\alpha(\boldsymbol{e})] = \sigma_0^\alpha(pe[\boldsymbol{e}])$. If this extension is not possible, an error is signaled to the caller.

The *extend_iso* function is passed a (*pv*, *rots*, *pe*) triplet, a vertex $\boldsymbol{v}_1'$ of $\boldsymbol{G}_1$, a vertex $\boldsymbol{v}_2'$ of $\boldsymbol{G}_2$ and a rotation r; it alters the given (*pv*,*rots*,*pe*) triple by adding a mapping of the vertex $\boldsymbol{v}_1'$ into vertex $\boldsymbol{v}_2'$ (and rotating the target vertex by r places rightwards). If the extension is successful, it returns the extended map (*pv*, *rot*, *pe*); otherwise, signals an error.

2.2. Operations with *Fatgraph* Isomorphisms.

Compare pull-back orientation. The *compare_orientations* function takes an isomorphism triple (*pv*, *rots*, *pe*) and a pair of *Fatgraph* objects $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, and returns $+1$ or -1 depending on whether the orientations of the target *Fatgraph* pulls back to the orientation of the source *Fatgraph* via the given isomorphism.

Recall that for a *Fatgraph* object $\boldsymbol{G}$, the orientation is represented by a mapping $\boldsymbol{G}.orient$ that associates an edge $\boldsymbol{e}$ with its position in the wedge product that represents the orientation; therefore, the pull-back orientation according to an isomorphism (*pv*, *rots*, *pe*) from $\boldsymbol{G}$ to $\boldsymbol{G}'$ is simply given by the map $\boldsymbol{e} \mapsto \boldsymbol{G}'.orient[pe[\boldsymbol{e}]]$. Thus, the comparison is done by constructing the permutation that maps $\boldsymbol{G}.orient[\boldsymbol{e}]$ to $\boldsymbol{G}'.orient[pe[\boldsymbol{e}]]$ and taking its sign (which has linear complexity with respect to the number of edges).

The is_orientation_reversing function. Determining whether an automorphism reverses orientation is crucial for knowing which fatgraphs are orientable. Function *is_orientation_reversing* takes a *Fatgraph* object and an isomorphism triple (*pv*, *rots*, *pe*) as input, and returns boolean *True* iff the isomorphism reverses orientation. This amounts to checking whether the given orientation and that of the pull-back one agree, which can be done with the comparison method discussed above.

Algorithm 6 Function *MgnGraphs* returns all connected fatgraphs having prescribed genus g and number of boundary cycles n. Actual output of the function is a list R, whose k-th element $R[k]$ is itself a list of graphs in $\mathcal{R}_{g,n}$ with $m-k$ edges.

```
1  def MgnGraphs(g,n):
2    m ← 4g + 2n − 5 maximum number of edges
3    R ← array of m empty lists
4    R[0] ← MgnTrivalentGraphs(g,n) first item contains all 3−valent graphs
5    for k in 1, \ldots, m − 1:
6      Initialize R[k] as an empty list
7      for G in R[k − 1]:
8        for e in edge_orbits(G):
9          if e is a loop:
10           continue with next e
11         G′ ← contract(G, e)
12         if G′ not already in R[k]:
13           append G′ to R[k]
14   return R
```

Transforming boundary cycles under an isomorphism. The function *transform_boundary_cycle* is used when comparing *marked* fatgraphs: as the marking is a function on the boundary cycles, we need to know exactly which boundary cycle of the target graph corresponds to a given boundary cycle in the source graph.

Recall that *BoundaryCycle* instances are defined as list of *corners*; function *transform_boundary_cycle* takes a *BoundaryCycle* $\boldsymbol{b}$ and returns a new *BoundaryCycle* object $\boldsymbol{b}'$, obtained by transforming each corner according to a graph isomorphism. Indeed, *transform_boundary_cycle* is straightforward loop over the corners making up $\boldsymbol{b}$: For each corner $(\boldsymbol{v},i,j)$, a new one is constructed by transforming the vertex according to map pv, and displacing indices i and j by the rotation amount indicated by $rot[\boldsymbol{v}]$ (modulo the number of edges attached to $\boldsymbol{v}$).

3. Generation of fatgraphs

Let *MgnGraphs* be the function which, given two integers g, n as input, returns the collection of $\mathcal{R}_{g,n}$ graphs. Let us further stipulate that the output result will be represented as a list R: the 0-th item in this list is the list of graphs with the maximal number m of edges; the k-th item $R[k]$ is the list of graphs having $m-k$ edges. There are algorithmic advantages in this subdivision, which are explained below.

Graphs with the maximal number of edges are trivalent graphs; they are computed by a separate function *MgnTrivalentGraphs*, described in Section 3.1.

We can then proceed to generate all graphs in $\mathcal{R}_{g,n}$ by contraction of regular edges: through contracting one edge in trivalent graphs we get the list $R[1]$ of all graphs with $m-1$ edges; contracting one edge of $G \in R[1]$, we get $G' \in R[2]$ with $m-2$ edges, and so on. Pseudo-code for *MgnGraphs* is shown in Algorithm 6. The loop at lines 8–13 is the core of the function: contract edges of the fatgraph G (with $m-k+1$ edges) to generate new fatgraphs with $m-k$ edges. However, we need not contract every edge of a fatgraph: if $a \in \operatorname{Aut} G$ is an automorphism and $e \in E(G)$

is an edge, then the contracted graphs $G' = G/e$ and $G'' = G/a(e)$ are isomorphic. Hence, we can restrict the computation to only one representative edge per orbit of the action induced by $\mathrm{Aut}\, G$ on the set $E(G)$; the *edge_orbits* function referenced at line 8 should return a list of representative edges, one per each orbit of $\mathrm{Aut}(G)$ on $E(G)$.

Lines 12–13 add $\boldsymbol{G'}$ to $R[k]$ *only if it is not already there.* This is the most computationally expensive part of the *MgnGraphs* function: we need to perform a comparison between $\boldsymbol{G'}$ and each element in $R[k]$; testing equality of two fatgraphs requires computing if there are isomorphisms between the two, which can only be done by attempting enumeration of such isomorphisms. (Fatgraph isomorphism is discussed in detail in Section 2.)

If N_k is the number of elements in $R[k]$ and T_{iso} is the average time needed to determine if two graphs are isomorphic, then evaluating whether $\boldsymbol{G'}$ is already contained in $R[k]$ takes $O(N_k \cdot T_{\mathrm{iso}})$ time: thus, the subdivision of the output R into lists, each one holding graphs with a specific number of edges, reduces the number of fatgraph comparisons done in the innermost loop of *MgnGraphs*, resulting in a substantial shortening of the total running time.

Note that the top-level function *MgnGraphs* is quite independent of the actual implementation of the *Fatgraph* type of objects: all is needed here, is that we have methods for enumerating edges of a *Fatgraph* object, contracting an edge, and testing two graphs for isomorphism.

Lemma 3.28. *If MgnTrivalentGraphs*(g, n) *returns the complete list of* trivalent *fatgraphs in* $\mathcal{R}_{g,n}$*, then the function MgnGraphs defined above returns the complete set of fatgraphs* $\mathcal{R}_{g,n}$*.*

Proof. By the above dissection of the algorithm, all we need to prove is that any fatgraph in $\mathcal{R}_{g,n}$ can be obtained by a chain of edge contractions from a trivalent fatgraph. This follows immediately from the fact that any fatgraph vertex v of valence $z \geqslant 3$ can be expanded (in several ways) into vertices v_1, v_2 of valences z_1, z_2 such that $z = (z_1 - 1) + (z_2 - 1)$, plus a connecting edge. □

3.1. Generation of Trivalent Fatgraphs. Generation of trivalent graphs can be tackled by an inductive procedure: given a trivalent graph, a new edge is added, which joins the midpoints of two existing edges. In order to determine which graphs should be input to this "edge addition" procedure, one can follow the reverse route, and ascertain how a trivalent graph is transformed by *deletion* of an edge.

Throughout this section, l and m stand for the number of vertices and edges of a graph; it will be clear from the context, which exact graph they are invariants of.

3.1.1. *Removal of edges.* Let $G \in \mathcal{R}_{g,n}$ be a *connected* trivalent graph. Each edge $x \in E(G)$ falls into one of the following categories:

- *A)* x is a loop: both endpoints of x are attached to a single vertex v, another edge x' joins v with a distinct vertex v';
- *B)* x joins two distinct vertices $v, v' \in V(G)$ and separates two distinct boundary cycles $\beta, \beta' \in B(G)$;
- *C)* x joins two distinct vertices $v, v' \in V(G)$ but belongs to only one boundary cycle $\beta \in B(G)$, within which it occurs twice (once for each orientation).

Deletion of edge x requires different adjustments in order to get a trivalent graph again in each of the three cases above; it also yields a different result in each case.

Case *A)*: If x is a loop attached to v, then, after deletion of x, one needs to also delete the loose edge x' and the vertex v' (that is, join the two other edges attached to v'; see Figure 3.8, bottom row). The resulting fatgraph G' has:

- » two vertices less than G: v and v' have been deleted;
- » three edges less: x, x' have been deleted and two other edges merged into one;
- » one boundary cycle less: the boundary cycle totally bounded by x has been removed.

Therefore:

$$\begin{aligned} 2-2g' = \chi(G') &= l' - m' + n' \\ &= (l-2) - (m-3) + (n-1) \\ &= l - m + n = \chi(G) = 2-2g, \end{aligned}$$

hence $g = g'$, and

$$G' \in \mathcal{R}_{g,n-1}. \tag{A}$$

Case *B)*: x joins distinct vertices v, v' and separates distinct boundary cycles (see Figure 3.8, top row). Delete x and merge the two edges attached to each of the two vertices v and v'; in the process, the two boundary cycles β, β' also merge into one. The resulting fatgraph G' is connected. Indeed, given any two vertices $u, u' \in V(G')$, there is a path $(x_1, \ldots, x_k)$ connecting u with u' in G. If this path passes through x, one can replace the occurrence of x with the perimeter — excluding x— of one of the two boundary cycles β, β' to get a path joining v and v' which avoids x, and thus projects to a path in G'. Again we see that G' has:

- » two vertices less than G: v and v' have been deleted;
- » three edges less: x has been deleted and four other edges merged into two, pair by pair;
- » one boundary cycle less: the boundary cycles β, β' have been merged into one.

Therefore $g = g'$, and

$$G' \in \mathcal{R}_{g,n-1}. \tag{B}$$

Case *C)*: x joins distinct vertices v, v' but belongs into *one* boundary cycle $\beta \in B(G)$ only. Delete edge x and the two vertices v, v', joining the attached edges two by two as in case *B)*. We distinguish two cases, depending on whether the resulting fatgraph is connected.

C') If the resulting fatgraph G' is connected, then $\beta \in B(G)$ has been split into two distinct boundary cycles $\beta', \beta'' \in B(G')$. Indeed, write the boundary cycle β as an ordered sequence of *oriented* edges: $y_0 \to y_1 \to \ldots \to y_k \to y_0$. Assume the y_* appear in this sequence in the exact order they are encountered when walking along β in the sense given by the fatgraph orientation. The oriented edges y_i are pairwise distinct: if y_i and y_j share the same supporting edge, then y_i and y_j have opposite orientations. By the initial assumption of case *C)*, edge x must appear *twice* in the list: if $\bar{x}$ and $\underline{x}$ denote the two orientations of x, then $y_i = \bar{x}$ and $y_j = \underline{x}$. Deleting x from β is (from a homotopy point of view) the same as replacing $y_i = \bar{x}$

with $\bar{x} \to \underline{x}$, and $y_j = \underline{x}$ with $\underline{x} \to \bar{x}$ when walking a boundary cycle. Then we see that β splits into two disjoint cycles:

$$\beta' = y_0 \to y_1 \to \cdots \to y_{i-1} \to \bar{x} \to \underline{x} \to y_{j+1} \to \cdots \to y_k \to y_0,$$
$$\beta'' = y_{i+1} \to \cdots \to y_{j-1} \to \underline{x} \to \bar{x} \to y_{i+1}.$$

In this case, G' has:

- » two vertices less than G: v and v' have been deleted;
- » three edges less: x has been deleted and four other edges merged into two, pair by pair;
- » one boundary cycle *more*: the boundary cycle β has been split in the pair β', β''.

Therefore $g' = g - 1$ and $n' = n + 1$, so:

$$G' \in \mathcal{R}_{g-1,n+1}. \tag{C'}$$

C") G' is a disconnected union of fatgraphs G'_1 and G'_2; for this statement to hold unconditionally, we temporarily allow a single circle into the set of connected fatgraphs (consider it a fatgraph with one closed edge and no vertices) as the one and only element of $\mathcal{R}_{0,2}$. As will be shown in Lemma 3.29, this is irrelevant for the *MgnTrivalentGraphs* algorithm. Now:

$$l'_1 + l'_2 = l - 2, \qquad m'_1 + m'_2 = m - 3, \qquad n'_1 + n'_2 = n + 1,$$

hence:

$$\begin{aligned}(2 - 2g'_1) + (2 - 2g'_2) &= (l - 2) - (m - 3) + (n + 1) \\ &= (l - m + n) + 2 = 4 - 2g\end{aligned}$$

So that $g'_1 + g'_2 = g + 2$, $n'_1 + n'_2 = n + 1$, and:[8]

$$G' = G'_1 \otimes G'_2 \in \mathcal{R}_{g'_1,n'_1} \otimes \mathcal{R}_{g'_2,n'_2}. \tag{C"}$$

3.1.2. *Inverse construction.* If $x \in E(G)$ is an edge of a fatgraph G, denote $\bar{x}$ and $\underline{x}$ the two opposite orientations of x.

In the following, let $\mathcal{R}'_{g,n}$ be the set of fatgraphs with a selected oriented edge:

$$\mathcal{R}'_{g,n} := \{(G, \bar{x}) : G \in \mathcal{R}_{g,n}, \bar{x} \in L(G)\}.$$

Similarly, let $\mathcal{R}''_{g,n}$ be the set of fatgraphs with two chosen oriented edges:

$$\mathcal{R}''_{g,n} := \{(G, \bar{x}, \bar{y}) : G \in \mathcal{R}_{g,n}, \bar{x}, \bar{y} \in L(G)\}.$$

The following abbreviations are convenient:

$$\mathcal{R} = \cup\mathcal{R}_{g,n}, \qquad \mathcal{R}' = \cup\mathcal{R}'_{g,n}, \qquad \mathcal{R}'' = \cup\mathcal{R}''_{g,n}.$$

Define the attachment of a new edge to a fatgraph in the following way. Given a fatgraph G and an *oriented* edge $\bar{x}$, we can create a new trivalent vertex v in the midpoint of x, and attach a new edge to it, in such a way that the two halves of x appear, in the cyclic order at v, in the same order induced by the orientation of $\bar{x}$. Figure 3.9 depicts the process.

We can now define maps that invert the constructions *A)*, *B)*, *C')* and *C")* defined in the previous section.

[8]Here we use $\otimes$ to indicate juxtaposition of graphs: $G_1 \otimes G_2$ is the (non-connected) fatgraph having two connected components G_1 and G_2.

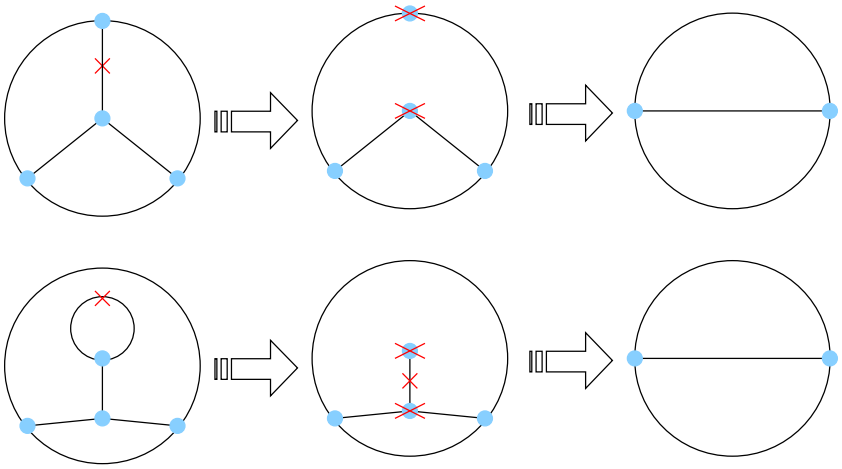


FIGURE 3.8. Graphical illustration of fatgraph edge removal. *Top row:* a regular edge (crossed) is removed from an $\mathcal{R}_{0,4}$ graph; its endpoints are further removed; the remaining edges are joined and the resulting graph is a trivalent fatgraph in $\mathcal{R}_{0,3}$. *Bottom row:* a loop is removed from a trivalent $\mathcal{R}_{0,4}$ graph; the stem together with its endpoints has to be removed as well; the remaining edges are joined, and we end up with a trivalent fatgraph in $\mathcal{R}_{0,3}$.

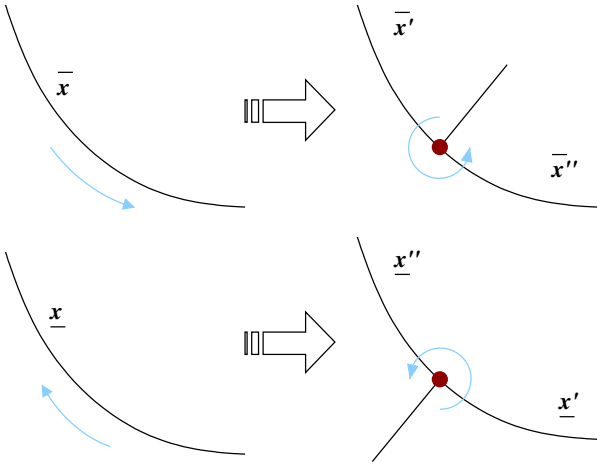


FIGURE 3.9. When adding a new vertex in the middle of an edge x, the cyclic order depends on the oriented edge: the two orientations $\bar{x}$ and $\underline{x}$ get two inequivalent cyclic orders.

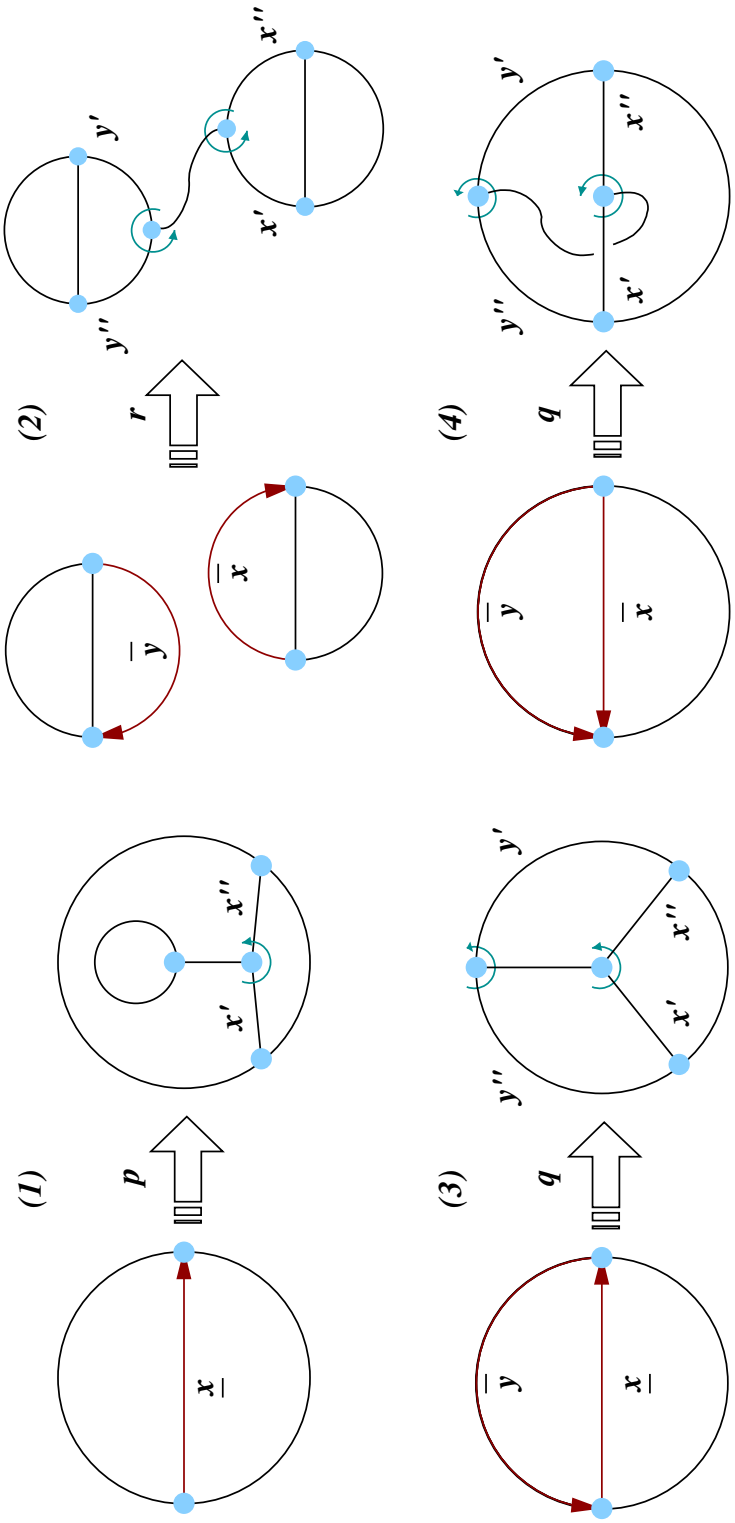


FIGURE 3.10. Graphical illustration of maps p, q, $r_{g,n}$. *(1):* $p(G, \bar{x})$ attaches a "slipknot" to edge $\bar{x}$. *(2):* $r_{2,5}(G_1, \bar{x}, G_2, \bar{y})$ joins fatgraphs G_1 and G_2 with a new edge. *(3) and (4):* it is shown how changing the orientation of an edge can lead to different results in *(3)* $q(G, \bar{x}, \bar{y})$ and *(4)* $p(G, \underline{x}, \bar{y})$.

Let $p_{g,n} : \mathcal{R}'_{g,n-1} \to \mathcal{R}_{g,n}$ be the map that creates a fatgraph $p(G, \bar{x})$ from a pair $(G, \bar{x})$ by attaching the loose end of a "slip knot"[9] to the midpoint of x. The map $p : \mathcal{R}' \to \mathcal{R}$ defined by $p|_{\mathcal{R}'_{g,n}} := p_{g,n}$ is ostensibly inverse to *A)*.

To invert *B)* and *C')*, define a map $q : \mathcal{R}'' \to \mathcal{R}$ that operates as follows:

- Given $(G, \bar{x}, \bar{y})$ with $\bar{x} \neq \bar{y}$, the map q attaches a new edge to the midpoints of x and y; again the cyclic order on the new midpoint vertices is chosen such that the two halves of x and y appear in the order induced by the orientations $\bar{x}$, $\bar{y}$.
- When $\bar{x} = \bar{y}$, let us further stipulate that the construction of $q(G, \bar{x}, \bar{x})$ happens in two steps:
  (1) a new trivalent vertex is created in the midpoint of $x \in E(G)$ and a new edge ξ is attached to it,
  (2) create a new trivalent vertex in the middle of the half-edge which comes first in the ordering induced by the orientation $\bar{x}$; attach the loose end of the new edge ξ to this new vertex.

  It is clear that the above steps give an unambiguous definition of q in all cases where $\bar{x}$ and $\bar{y}$ are orientations of the same edge of G, that is, $(G, \bar{x}, \bar{x})$, $(G, \bar{x}, \underline{x})$, $(G, \underline{x}, \bar{x})$, and $(G, \underline{x}, \underline{x})$.

Ostensibly, q inverts the edge removal in cases *B)* and *C')*: the former applies when a graph $G \in \mathcal{R}_{g,n}$ is sent to $q(G) \in \mathcal{R}_{g,n+1}$, the latter when $G \in \mathcal{R}_{g,n}$ is sent to $q(G) \in \mathcal{R}_{g+1,n-1}$.

Finally, to invert *C")*, let us define

$$r_{g,n} : \bigoplus_{\substack{g'_1+g'_2=g+2 \\ n'_1+n'_2=n}} \mathcal{R}'_{g'_1,n'_1} \times \mathcal{R}'_{g'_2,n'_2} \to \mathcal{R}.$$

From $(G', \bar{x}', G'', \bar{x}'')$, construct a new fatgraph by bridging G' and G'' with a new edge, whose endpoints are in the midpoints of x' and x''; again, stipulate that the cyclic order on the new vertices is chosen such that the two halves of x', x'' appear in the order induced by the orientations $\bar{x}'$, $\bar{x}''$.

Summing up, any fatgraph $G \in \mathcal{R}_{g,n}$ belongs to the image of one of the above maps p, q, and r. There is considerable overlap among the different image sets: in fact, one can prove that r is superfluous.

Lemma 3.29. *Any fatgraph obtained by inverting construction C") lies in the image of maps p and q.*

Proof. Assume, on the contrary, that G lies in the image of r only. Then, deletion of any edge x from G yields a disconnected graph $G' \otimes G''$. Both subgraphs G' and G'' enjoy the same property, namely, that deletion of any edge disconnects: otherwise, if the removal of $y \in E(G')$ does not disconnect G', then neither does it disconnect $G = r_{g,n}(G', G'')$, contrary to the initial assumption. As long as G' or G'' has more than 3 edges, we can delete another edge; by recursively repeating the process, we end up with a fatgraph G^* with $l^* \leqslant 3$ edges, which is again disconnected by removal of any edge. Since G^* is trivalent, $3 \cdot m^* = 2 \cdot l^*$, therefore G^* must have exactly 3 edges and 2 vertices. But all such fatgraphs belong to $\mathcal{R}_{0,3}$ or $\mathcal{R}_{1,1}$, and it is readily checked that there is no way to add an edge such that the required property holds, that any deletion disconnects. □

[9] A single 3-valent vertex with one loop attached and a regular edge with one loose end.

3.1.3. *The MgnTrivalentGraphs algorithm.* The stage is now set for implementing the recursive generation of trivalent graphs. Pseudo-code is listed in Algorithm 7.

Lemma 3.30. *MgnTrivalentGraphs*(g,n) *generates all trivalent fatgraphs for each given* g, n. *Only one representative per isomorphism class is returned.*

PROOF. The function call *MgnTrivalentGraphs*(g,n) recursively calls itself to enumerate trivalent graphs of $\mathcal{R}_{g,n-1}$ and $\mathcal{R}_{g-1,n+1}$. In particular, *MgnTrivalentGraphs* must:

» provide the full set of fatgraphs $\mathcal{R}_{0,3}$ and $\mathcal{R}_{1,1}$ as induction base.
» return the empty set when called with an invalid (g,n) pair;

The general case is then quite straightforward: *(1)* apply maps p, q to every fatgraph in $\mathcal{R}_{g,n-1}$, and q to every fatgraph in $\mathcal{R}_{g-1,n+1}$; *(2)* discard all graphs that do not belong to $\mathcal{R}_{g,n}$; and *(3)* take only one graph per isomorphism class into the result set.

To invert construction *A)*, map p is applied to all fatgraphs $G \in \mathcal{R}_{g,n-1}$; if $a \in \operatorname{Aut} G$, then $p(a(G), a(x)) = p(G,x)$, therefore we can limit ourselves to one pair (G,x) per orbit of the automorphism group, saving a few computational cycles. Similarly, since q is a function of $(G,\bar{x},\bar{y})$, which is by construction invariant under $\operatorname{Aut} G$, we can again restrict to considering only one $(G,\bar{x},\bar{y})$ per $\operatorname{Aut} G$-orbit; this is computed by the *edge_pair_orbits*(G) function. □

Note that there is no way to tell from G if fatgraphs $p(G,x)$ and $q(G,x,y)$ belong to $\mathcal{R}_{g,n}$: one needs to check g and n before adding the resulting fatgraph to the result set R.

The selection of only one representative fatgraph per isomorphism class can be done by removing duplicates from the collection of generated graphs in the end, or by running the isomorphism test before adding each graph to the working list R. The computational complexity is quadratic in the number of generated graphs in both cases, but the latter option requires less memory. In any case, this isomorphism test is the most computationally intensive part of *MgnTrivalentGraphs*.

For an expanded discussion of the size of the result set R, and a comparison with other generation algorithms, see Appendix B. It would be interesting to reimplement the trivalent generation algorithm using the technique outlined in [47], and compare it with the current (rather naive) algorithm.

3.1.4. *Implementing maps* $p(G,x)$ *and* $q(G,x,y)$. Implementation of both functions is straightforward and pseudo-code is therefore omitted;[10] the only question is how to represent the "oriented edges" that appear in the signature of maps p and q.

In both p and q, the oriented edge $\bar{x}$ or $\underline{x}$ is used to determine how to attach a new edge to the midpoint of the target (unoriented) edge x. We can thus represent an oriented edge as a pair $(\boldsymbol{e}, s)$ formed by a *Fatgraph* edge $\boldsymbol{e}$ and a "side" s: valid values for s are $+1$ and -1, interpreted as follows. The parameter s controls which of the two inequivalent cyclic orders the new trivalent vertex will be given. Let $\boldsymbol{a}$, $\boldsymbol{b}$, $\boldsymbol{c}$ be the edges attached to the new vertex in the middle of $\boldsymbol{e}$, where $\boldsymbol{a}$,$\boldsymbol{b}$ are the two halves of $\boldsymbol{e}$. If s is $+1$, then the new trivalent vertex will have the cyclic order $\boldsymbol{a} < \boldsymbol{b} < \boldsymbol{c} < \boldsymbol{a}$; if s_1 is -1, then the edges $\boldsymbol{a}$ and $\boldsymbol{b}$ are swapped and the new trivalent vertex gets the cyclic order $\boldsymbol{b} < \boldsymbol{a} < \boldsymbol{c} < \boldsymbol{b}$ instead.

[10] The interested reader is referred to the publicly-available code at http://fatghol.googlecode.com for details.

Algorithm 7 Return a list of all connected trivalent fatgraphs with prescribed genus g and number of boundary cycles n. A fatgraph if "admissible" iff it has the prescribed genus g and number of boundary cycles n.

```
def MgnTrivalentGraphs(g, n):
    avoid infinite recursion in later statements
  if n= 0 or (g,n) < (0,3):
    return empty list

   Induction base: M_{0,3} and M_{1,1}
  if (g,n) = (0,3):
    return list of fatgraphs in R_{0,3}
  elif (g,n) = (1,1):
    return list of fatgraphs in R_{1,1}

   general case
  else:
    R ← empty list

     case A): hang a circle to all edges of graphs in M_{g,n-1}
    for G in MgnTrivalentGraphs(g, n-1):
      for x in edge_orbits(G):
        add p(G, x̄) to R if admissible
        add p(G, x) to R if admissible

     case B): bridge all edges of a single graph in M_{g,n-1}
    for G in MgnTrivalentGraphs(g, n-1):
      for (x,y) in edge_pair_orbits(G):
        add q(G, x, y) to R if admissible
        add q(G, x, ȳ)) to R if admissible
        add q(G, x̄, y) to R if admissible
        add q(G, x̄, ȳ) to R if admissible

     case C'): bridge all edges of a single graph in M_{g-1,n+1}
    for G in MgnTrivalentGraphs(g-1, n+1):
      for (x,y) in edge_pair_orbits(G):
        add q(G, x, y) to R if admissible
        add q(G, x, ȳ) to R if admissible
        add q(G, x̄, y) to R if admissible
        add q(G, x̄, ȳ) to R if admissible

    remove isomorphs from R
    return R
```

4. The homology complex of marked fatgraphs

Betti numbers of a complex (W_*, D_*) can be reckoned (via a little linear algebra) from the matrix form $\boldsymbol{D}^{(k)}$ of the boundary operators D_k. Indeed, given that $b_k := \dim H_k(W, D)$ and $H_k(W, D) := Z_k(W, D)/B_k(W, D) = \operatorname{Ker} D_k/D_{k-1}(W_{k-1})$, by the rank-nullity theorem we have $\dim \operatorname{Ker} D_k = \dim W_k - \operatorname{rank} \boldsymbol{D}^{(k)}$ hence $b_k = \dim \operatorname{Ker} D_k - \dim D_{k-1}(W_{k-1}) = \dim W_k - \operatorname{rank} \boldsymbol{D}^{(k)} - \operatorname{rank} \boldsymbol{D}^{(k-1)}$.

In order to compute the matrix $\boldsymbol{D}^{(k)}$, we need to compute the coordinate vector of $D_k x_j^{(k)}$ for all vectors $x_j^{(k)}$ in a basis of W_k. If (W_*, D_*) is the fatgraph complex, then the basis vectors $x_j^{(k)}$ are marked fatgraphs with k edges, and the differential D_k is defined as an alternating sum of edge contractions. Therefore, in order to compute the coordinate vector of $D_k x_j^{(k)}$, one has to find the unique fatgraph $x_h^{(k-1)}$ which is isomorphic to a given contraction of $x_j^{(k)}$ and score a ± 1 coefficient depending on whether orientations agree or not.

Although this approach works perfectly, it is inefficient in practice. Indeed, lookups into the basis set $\{x_{h=1,\dots,N}^{(k-1)}\}$ of W_{k-1} require on average $O(N^2)$ isomorphism checks. Still, we can take a shortcut: if two topological fatgraphs G and G' are not isomorphic, so are any two marked fatgraphs (G, ν) and (G', ν'). Indeed, rearrange the rows and columns of the boundary operator matrix $\boldsymbol{D}^{(k)}$ so that marked fatgraphs (G, ν) over the same topological fatgraph G correspond to a block of consecutive indices. Then there is a rectangular portion of $\boldsymbol{D}^{(k)}$ that is uniquely determined by a pair of topological fatgraphs G and G'. The main function for computing the boundary operator matrix can thus loop over pairs of topological fatgraphs, and delegate computing the each rectangular block to specialized code. There are $n!/|\operatorname{Aut} G|$ marked fatgraphs per given topological fatgraph G, so this approach can cut running time down by $O((n!)^2)$.

The generation of inequivalent marked fatgraphs (over the same topological fatgraph G) can be reduced to the (computationally easier) combinatorial problem of finding cosets of a subgroup of the symmetric group $\mathfrak{S}_n$. In addition, the list of isomorphisms between G and G' can be cached and re-used for comparing all pairs of marked fatgraphs (G, ν), (G', ν'). This strategy is implemented by two linked algorithms:

(1) *MarkedFatgraphPool*: Generate all inequivalent markings of a given topological fatgraph G.
(2) *compute_block*: Given topological fatgraphs G and G', compute the rectangular block of a boundary operator matrix whose entries correspond to coordinates of $D(G, \nu)$ w.r.t. (G', ν').

4.1. Generation of inequivalent marked fatgraphs. For any marked fatgraph (G, ν), denote $[G, \nu]$ its isomorphism class; recall that $B(-)$ is the functor associating a fatgraph with the set of its boundary cycles. Let $N(G)$ be the sets of all markings over G and $\tilde{N}(G)$ the set of isomorphism classes thereof:

$$N(G) := \{\, (G, \nu) \mid \nu \colon B(G) \to \{1, \dots, n\} \,\},$$
$$\tilde{N}(G) := \{\, [G, \nu] \mid \nu \colon B(G) \to \{1, \dots, n\} \,\}.$$

Let $(G, \bar{\nu})$ be a chosen marked fatgraph. Define a group homomorphism:

$$\Phi : \operatorname{Aut}(G) \ni a \longmapsto \bar{\nu} \circ B(a) \circ \bar{\nu}^{-1} \in \mathfrak{S}_n. \tag{4.1}$$

The set $P = \Phi(\operatorname{Aut} G)$ is a subgroup of $\mathfrak{S}_n$.

Lemma 3.31. *The marked fatgraphs $(G, \bar{\nu})$ and $(G, \sigma\bar{\nu})$ are isomorphic if and only if $\sigma \in P$.*

PROOF. Let $\sigma \in P$, then $\sigma^{-1} \in P$ and there exists $a \in \operatorname{Aut} G$ such that:

$$\sigma^{-1} = \bar{\nu} \circ B(a) \circ \bar{\nu}^{-1},$$

whence:

$$(\sigma \circ \bar{\nu}) \circ B(a) \circ \bar{\nu}^{-1} = \mathrm{id},$$

therefore a induces a marked fatgraph isomorphism between $(G, \bar{\nu})$ and $(G, \sigma \circ \bar{\nu})$.

Conversely, let $\hat{\nu} = \sigma\bar{\nu}$ and assume $(G, \bar{\nu})$ and $(G, \hat{\nu})$ are isomorphic as marked fatgraphs: then there exists $a \in \operatorname{Aut} G$ such that $\hat{\nu} \circ B(a) \circ \bar{\nu}^{-1}$ is the identity. Given any $\bar{\nu} \circ B(a') \circ \bar{\nu}^{-1} \in P$ we have:

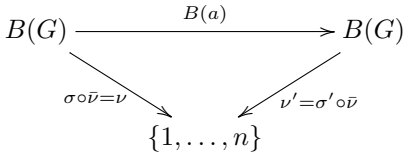
$$\begin{aligned} P \ni \bar{\nu} \circ B(a') \circ \bar{\nu}^{-1} &= \bar{\nu} \circ (\hat{\nu}^{-1} \circ \hat{\nu}) \circ B(a) \circ B(a)^{-1} \circ B(a') \circ \bar{\nu}^{-1} \\ &= (\bar{\nu} \circ \hat{\nu}^{-1}) \circ \hat{\nu} \circ B(a) \circ (\bar{\nu}^{-1} \circ \bar{\nu}) \circ B(a^{-1}) \circ B(a') \circ \bar{\nu}^{-1} \\ &= \sigma^{-1} \circ (\hat{\nu} \circ B(a) \circ \bar{\nu}^{-1}) \circ \bar{\nu} \circ B(a^{-1} \circ a') \circ \bar{\nu}^{-1} \\ &= \sigma^{-1} \circ \left(\bar{\nu} \circ B(a^{-1} \circ a') \circ \bar{\nu}^{-1}\right) \in \sigma^{-1} P, \end{aligned}$$

therefore $P = \sigma^{-1} P$, so $\sigma \in P$. □

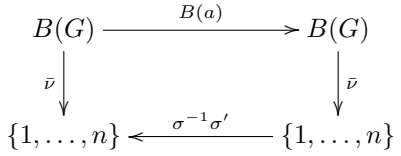
Define a transitive action of $\mathfrak{S}_n$ over $N(G)$ by $\sigma \cdot (G, \nu) := (G, \sigma\nu)$; this descends to a transitive action of $\mathfrak{S}_n$ on $\tilde{N}(G)$. By the previous Lemma, P is the stabilizer of $[G, \bar{\nu}]$ in $\tilde{N}(G)$.

Lemma 3.32. *The action of $\mathfrak{S}_n$ on $\tilde{N}(G)$ induces a bijective correspondence between isomorphism classes of marked fatgraphs and cosets of P in $\mathfrak{S}_n$.*

PROOF. Given isomorphic marked fatgraphs (G, ν) and (G, ν'), let $\sigma, \sigma' \in \mathfrak{S}_n$ be such that $\nu = \sigma \circ \bar{\nu}$ and $\nu' = \sigma' \circ \bar{\nu}$. By definition of marked fatgraph isomorphism, there is $a \in \operatorname{Aut} G$ such that the following diagram commutes:



Hence commutativity of another diagram follows:



Thus $(G, \bar{\nu})$ is isomorphic to $(G, \sigma^{-1}\sigma' \circ \bar{\nu})$; therefore $\sigma^{-1}\sigma' \in P$, hence $\sigma' \in \sigma P$, i.e., σ and σ' belong into the same coset of P.

Conversely, let $\tau, \tau' \in \sigma P$; explicitly:

$$\tau = \sigma \circ \bar{\nu} \circ B(a) \circ \bar{\nu}^{-1}, \qquad \tau' = \sigma \circ \bar{\nu} \circ B(a') \circ \bar{\nu}^{-1}.$$

Set $\nu = \tau \circ \bar{\nu}$, $\nu' = \tau' \circ \bar{\nu}$; substituting back the definition of τ, we have:

$$\nu = \sigma \circ \bar{\nu} \circ B(a) \circ \bar{\nu}^{-1} \circ \bar{\nu} = \sigma \circ \bar{\nu} \circ B(a),$$

whence $\bar{\nu} = \sigma^{-1} \circ \nu \circ B(a)^{-1}$, and:

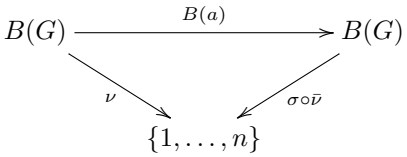
$$\nu' = \sigma \circ \bar{\nu} \circ B(a') = \sigma \circ \left(\sigma^{-1} \circ \nu \circ B(a)^{-1}\right) \circ B(a') = \nu \circ B(a^{-1} \circ a'),$$

therefore $a^{-1} \circ a'$ is an isomorphism between the marked fatgraphs (G, ν) and (G, ν'). □

The following is an easy corollary of the transitivity of the action of $\mathfrak{S}_n$ on $\tilde{N}(G)$.

Lemma 3.33. *Given any marking ν on the fatgraph G, there exist $\sigma \in \mathfrak{S}_n$ and $a \in \operatorname{Aut} G$ such that: $\nu = \sigma \circ \bar{\nu} \circ a$.*

Proof. By Lemma 3.32, there exists $\sigma \in \mathfrak{S}_n$ such that $[G, \nu] = [G, \sigma \circ \bar{\nu}]$, i.e., (G, ν) is isomorphic to $(G, \sigma \circ \bar{\nu})$. If $a \in \operatorname{Aut} G$ is this fatgraph isomorphism, then the following diagram commutes:



Therefore $\nu = \sigma \circ \bar{\nu} \circ B(a)$. □

The MarkedFatgraphPool algorithm. Given a fatgraph G and a *Fatgraph* object $\boldsymbol{G}$ representing it, let us stipulate that $\bar{\nu}$ be the marking on G that enumerates boundary cycles of G in the order they are returned by the function *compute_boundary_cycles*($\boldsymbol{G}$). By Lemma 3.33, every $(G, \nu^{(j)})$ can then be expressed (up to isomorphism) as $(G, \sigma^{(j)} \circ \bar{\nu})$ with $\sigma^{(j)} \in \mathfrak{S}_n$. The set $\{\sigma^{(j)}\}$ enumerates all distinct isomorphism classes of marked fatgraphs over G iff $\{\sigma^{(j)} P\}$ runs over all distinct cosets of P in $\mathfrak{S}_n$ (by Lemma 3.32).

The *MarkedFatgraphPool* function computes the set $\tilde{N}(G)$ of isomorphism classes $[G, \nu]$.

Theorem 3.34. *Given a Fatgraph $\boldsymbol{G}$ as input, MarkedFatgraphPool($\boldsymbol{G}$), as computed by Algorithm 8, outputs a tuple* (graph, P, A, markings, orientable)*, whose components are defined as follows:*

- *The* graph *item is the underlying* Fatgraph *object $\boldsymbol{G}$.*
- *The P slot holds a list of all elements in the group $P = \Phi(\operatorname{Aut} G)$.*
- *A stores a corresponding set of pre-image representatives (each element is an automorphism of $\boldsymbol{G}$): permutation $P[i]$ is induced by automorphism $A[i]$, i.e., if $\pi = P[i]$ and $a = A[i]$ then $\pi = \Phi(a)$.*
- *The* markings *item holds the list $\{\sigma^{(j)}\}$ of distinct cosets of P (representing inequivalent markings).*
- *orientable is a boolean value indicating whether any (G, ν) in the pool is orientable.*[11]

We need a separate boolean variable to record the orientability of the family of marked fatgraphs $N(G) = \{(G, \nu)\}$, because the automorphism group of a marked fatgraph $\operatorname{Aut}(G, \nu)$ can be a proper subgroup of $\operatorname{Aut} G$: hence, (G, ν) can be orientable even if G is not.

[11] It is an immediate corollary of Lemma 3.33 that if *one* marked fatgraph (G, ν^*) has an orientation-reversing automorphisms, then *every* marked fatgraph (G, ν) over the same topological fatgraph G has an orientation-reversing automorphism.

Algorithm 8 Compute the distinct markings of a given fatgraph. Input to the algorithm is a *Fatgraph* object $\boldsymbol{G}$; final result is a tuple *(*$\boldsymbol{G}$*, P, A, markings, orientable)* which represents the set $\tilde{N}(G)$ of isomorphism classes of marked fatgraphs.

```
def phi(a, G):
  π ← array of n elements
  for src_index, src_cycle in enumerate(G.boundary_cycles):
    dst_cycle ← a.transform_boundary_cycle(src_cycle)
    if dst_cycle not in G.boundary_cycles:
      abort and signal error to caller
    else:
      dst_index ← index of dst_cycle in G.boundary_cycles
    π[src_index] ← dst_index
  return π

def MarkedFatgraphPool(G):
  P ← empty list
  A ← empty list
   assume (G, ν) is orientable until we have counter−evidence
  orientable ← True
   step (1): loop over Aut G
  for a in G.automorphisms():
    try:
      π ← phi(a, G)
    except phi failed:
      continue with next a
    if permutation π is identity:
       found a new automorphism:
         − does it reverse orientation?
      if a.is_orientation_reversing():
        orientable ← False
         − does it define a new marking?
      if π not in P:
        append π to P
        append a to A
   step (2): enumerate cosets of P
  markings ← [ ]
  for σ in 𝔖_n:
    for π in P:
      if π ∘ σ in markings:
        continue with next σ
      add σ to markings
```

PROOF. Generation of all inequivalent markings over G is a direct application of Lemma 3.32, performed in two steps:

(1) In the first step: for each automorphism $a \in \operatorname{Aut} G$, compute the permutation $\Phi(a)$ it induces on the set of boundary components and form the subgroup P. The subgroup P and the associated set of automorphisms $A \subseteq \operatorname{Aut} G$ are stored in variables P and A.
(2) In the second step: compute cosets of P by exhaustive enumeration. They are recast into the list $\{\sigma^{(j)}\}$, which is stored into the *markings* variable.

As an important by-product of the computation, the automorphism group $\operatorname{Aut}(G, \bar{\nu})$ is computed, and used to determine if the marked fatgraphs in the pool are orientable.

The auxiliary function *phi* computes the permutation $\Phi(a) = \bar{\nu} \circ B(a) \circ \bar{\nu}^{-1}$. A permutation π is created and returned; it is represented by an array with n slots, which is initially empty and is then stepwise constructed by iterating over boundary cycles. Indeed, the boundary cycle *src_cycle* is transformed according to $B(a)$ and its position in the list of boundary cycles of $\boldsymbol{G}$ is then looked up. Note that this lookup may fail: there are in fact cases, in which the *Fatgraph.isomorphisms* algorithm finds a valid mapping, that however does not preserve the markings on boundary cycles; such failures need to be dealt with by rejecting a.

Step *(1)* of the computation is performed in lines 18–27:

» Computation of the permutation π (induced by a on the boundary cycles of $\boldsymbol{G}$) may fail; if this happens, the algorithm ignores a and proceeds with another automorphism.
» If a preserves the boundary cycles, then it induces an automorphism of the marked graph and we need to test whether it preserves or reverses orientation.
» There are $|\operatorname{Ker} \Phi|$ distinct automorphisms inducing the same permutation on boundary cycles: if π is already in P, discard it and continue with the next a.

By Lemma 3.32, there are as many distinct markings as there are cosets of P in $\mathfrak{S}_n$. Step *(2)* of the algorithm proceeds by simply enumerating all permutations in $\mathfrak{S}_n$, with *marking* initially set to the empty list; for each permutation σ a test is made as to whether σP intersects the list *markings* (lines 35–37); if it does not, then the marking induced by σ is added to the list.

□

A constructive version of Lemma 3.33 can now be implemented: the following function *index_and_aut*, given a *Fatgraph* object $\boldsymbol{G}$ and a marking, returns the permutation (by index number j in $\boldsymbol{G}$.*markings*) and fatgraph automorphism $a = \boldsymbol{G}.A[i]$ such that the topological fatgraph $\boldsymbol{G}$ decorated with *marking* is isomorphic (through a) to the same graph decorated with $\boldsymbol{G}$.*markings*$[j]$.

```
1  def index_and_aut(G, marking):
2    for (i, π) in enumerate(G.P):
3      τ ← σ ∘ π
4      if τ in G.markings:
```

```
        j ← index of τ in G.markings
        return (j, G.A[i])
    else:
        continue with next π
```

The algorithm enumerates all permutations $\pi \in P$, and compares $\sigma \circ \pi$ to every element of $\boldsymbol{G}.markings$: by Lemma 3.32, we know that one must match.

4.2. Computing boundary operator matrix blocks. The differential $D(G, \nu)$ is computed by summing contractions of regular edges in G (with alternating signs); likewise, the matrix block corresponding to coordinates of the families of marked fatgraphs $\{(G, \nu)\}$ and $\{(G', \nu')\}$ can be decomposed into a sum of blocks, each block representing the coordinates of $\{(G/e, \nu)\}_{e \in E(G)}$ projected on the linear span of $\{(G', \nu')\}$.

More precisely, given any two fatgraphs G_1 (with m edges) and G_2 (with $m-1$ edges), let $X_1, X_2 \subseteq \mathscr{R}_{g,n}$ be the linear span of $N(G_1)$ and $N(G_2)$ respectively, and denote by pr_{X_2} the linear projection on subspace X_2. Recall that, for any fatgraph G, we have $D(G) = \sum \pm d^{(e)}(G)$, where the sum is taken over all regular edges e of G, and $d^{(e)}$ is the contraction of edge e.

Let G be the fatgraph obtained by contracting the chosen edge e in G_1. If G_2 and G are isomorphic, then the three graphs are related by the following diagram of fatgraph morphisms, where f_1 is the contraction map and f_2 is a fatgraph isomorphism:

$$\begin{array}{ccc} G_1 & & \\ {\scriptstyle f_1}\downarrow & & \\ G & \xrightarrow[f_2]{\sim} & G_2 \end{array} \tag{4.2}$$

The above diagram (4.2) functorially induces a diagram on the set of boundary cycles:

$$\begin{array}{ccc} B(G_1) & \xrightarrow{\nu_1} & \{1, \ldots, n\} \\ {\scriptstyle B(f_1)}\downarrow = & \nearrow_{\nu} & \uparrow{\scriptstyle \nu_2} \\ B(G) & \xrightarrow[B(f_2)]{\sim} & B(G_2) \end{array} \tag{4.3}$$

Diagram (4.3) commutes iff f_1, f_2 can be extended to morphisms of marked fatgraphs $\hat{f}_1 : (G_1, \nu_1) \to (G, \nu)$ and $\hat{f}_2 : (G, \nu) \to (G_2, \nu_2)$.

Now choose *Fatgraph* objects $\boldsymbol{G}_1$, $\boldsymbol{G}$, $\boldsymbol{G}_2$ representing G_1, G, G_2.

Let $\bar{\nu}_1$, $\bar{\nu}$, $\bar{\nu}_2$ be the markings on G_1, G, G_2 that enumerate boundary cycles in the order they are returned by the function *compute_boundary_cycle* applied to $\boldsymbol{G}_1$, $\boldsymbol{G}$, $\boldsymbol{G}_2$ respectively. Define $\phi_1, \phi_2 \in \mathfrak{S}_n$ by:

$$\phi_1 := \bar{\nu} \circ B(f_1) \circ \bar{\nu}_1^{-1}, \qquad \phi_2 := \bar{\nu}_2 \circ B(f_2) \circ \bar{\nu}^{-1}. \tag{4.4}$$

Lemma 3.35. *Given any marking ν_1 on G_1, choose $\sigma_1 \in \mathfrak{S}_n$ such that $\nu_1 = \sigma_1 \circ \bar{\nu}_1$ and define:*

$$\nu_2 := \sigma_1 \circ \phi_1^{-1} \circ \phi_2^{-1} \circ \bar{\nu}_2, \tag{4.5}$$

Then ν_2 is the unique marking on G_2 such that diagram (4.3) *commutes.*

PROOF. Let $\sigma_2 := \sigma_1 \circ \phi_1^{-1} \circ \phi_2^{-1}$. We need to prove that the external square in diagram (4.3) is commutative; indeed, we have:

$$\sigma_2 = \sigma_1 \circ (\bar\nu_1 \circ B(f_1)^{-1} \circ \bar\nu^{-1}) \circ (\bar\nu \circ B(f_2)^{-1} \circ \bar\nu_2) \\ = \sigma_1\bar\nu_1 \circ B(f_2 \circ f_1)^{-1} \circ \bar\nu_2^{-1},$$

so that:

$$\nu_2 \circ B(f_2) \circ B(f_1) = \sigma_2 \circ \nu_2 \circ B(f_2 \circ f_1) \\ = \sigma_1\bar\nu_1 \circ B(f_2 \circ f_1)^{-1} \circ \bar\nu_2^{-1} \circ \nu_2 \circ B(f_2 \circ f_1) \\ = \sigma_1 \circ \bar\nu_1 = \nu_1.$$

The uniqueness assertion is of immediate proof, since maps $B(f_1)$ and $B(f_2)$ are invertible. □

Let $\boldsymbol{p}_1$, $\boldsymbol{p}_2$ be the *MarkedFatgraphPool* output corresponding to $\boldsymbol{G}_1$, $\boldsymbol{G}_2$, and let $\{\nu_1^{(j)}\}_{j=1,\dots,N_1}$, $\{\nu_2^{(k)}\}_{k=1,\dots,N_2}$ be the enumeration of fatgraph markings corresponding to items in the lists $\boldsymbol{p}_1$.*markings* and $\boldsymbol{p}_2$.*markings* respectively.

Lemma 3.36. *For any regular edge e of G_1, and any choice of $j \in \{1, \dots, N_1\}$, there exist unique $k \in \{1, \dots, N_2\}$ and $s \in \{-1, 0, +1\}$ such that:*

$$\mathrm{pr}_{X_2}\left(d^{(e)}[G_1, \nu_1^{(j)}]\right) = s \cdot [G_2, \nu_2^{(k)}]. \tag{4.6}$$

PROOF. If G_2 and $G = G_1/e$ are not isomorphic, then, for any marking ν_1, $d(G_1, \nu_1)$ has no component in the subspace $X_2 = \{(G_2, \nu_2)\}$, so the assertion is true with $s = 0$.

Otherwise, by Lemma 3.35, given $\nu_1 = \nu_1^{(j)}$ there is a unique ν_2 such that s can be non-null; by Lemma 3.33, there exist $\nu_2^{(k)} := \sigma_2^{(k)} \circ \bar\nu_2$ and $a \in \mathrm{Aut}\, G$ such that:

(1) the marked fatgraph $(G_2, \nu_2^{(k)})$ is a representative of the isomorphism class $[G_2, \nu_2]$;
(2) a gives the isomorphism between marked fatgraphs (G_2, ν_2) and $(G_2, \nu_2^{(k)})$;
(3) $\nu_2^{(k)}$ is the marking on G_2 represented by k-th item in list $\boldsymbol{p}_2$.*markings*.

The coefficient s must then be ± 1 since both $(G_2, \nu_2^{(k)})$ and $d^{(e)}(G_1, \nu_1^{(j)})$ are (isomorphic to) elements in the basis of X_2. □

Theorem 3.37. *Given MarkedFatgraphPool objects $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and a chosen edge $\boldsymbol{e}$ of $\boldsymbol{G}_1$, the function compute_block in Algorithm 9 returns the set S of all triplets (j, k, s) with $s = \pm 1$ such that:*

$$\mathrm{pr}_{X_2}\left(d^{(e)}[G_1, \nu_1^{(j)}]\right) = s \cdot [G_2, \nu_2^{(k)}]. \tag{4.7}$$

PROOF. The algorithm closely follows the computation done before Lemma 3.35 and in the proof of Lemma 3.36.

If G_2 and $G = G_1/e$ are not isomorphic, then $d^{(e)}[G_1, \nu_1]$ has no component in the subspace X_2 generated by $\{[G_2, \nu_2]\}$, whatever the marking ν_1. The assertion is thus satisfied by $S = \varnothing$, i.e., an empty list of triplets (j, k, s) (lines 5–6 in Algorithm 9).

If G_2 is isomorphic to $G = G_1/e$ through f_2, then Lemma 3.35 provides the explicit formula $\nu_2^{(k)} = \sigma_1^{(j)} \circ \phi_1^{-1} \circ \phi_2^{-1} \circ \bar\nu_2$, where $\sigma_1^{(j)} = \nu_1^{(j)} \circ \bar\nu_1^{-1}$.

Algorithm 9 Return the set S of triplets (j, k, s) such that eq. (4.7) holds for $(G_1, \nu_1^{(j)})$ and $(G_2, \nu_2^{(k)})$ obtained by contracting $\boldsymbol{e}$ in all marked graphs in $\boldsymbol{p}_1$ and projecting onto graphs in the $\boldsymbol{p}_2$ family.

```
def compute_block(p_1, e, p_2):
  G_1 ← p_1.graph
  G_2 ← p_2.graph
  G ← contract(G_1, e)
  if G and G_2 are not isomorphic:
    return empty list
  else:
    result ← empty list
    f_2 ← first isomorphism computed by Fatgraph.isomorphisms(G, G_2)
    φ_1^{-1} ← compute_phi1_inv(G, G_1, e)
    φ_2^{-1} ← compute_phi2_inv(G, G_2, f_2)
    for (j, σ) in enumerate(p_1.markings):
      (k, a) ← index_and_aut(p_2, σ ∘ φ_1^{-1} ∘ φ_2^{-1})
      p ← G_1.orient[e]
      s ← (-1)^p * compare_orientations(f_2) * compare_orientations(a)
      append (j, k, s) to result
    return result

def compute_phi1_inv(G, G_1, e):
  τ ← empty array of n elements
  for i, b in enumerate(G_1.boundary_cycles):
    b' ← contract_boundary_cycle(G_1, b, e)
    i' ← index of b' in G.boundary_cycles
    τ[i'] ← i
  return τ

def compute_phi2_inv(G, G_2, f_2):
  τ' ← empty array of n elements
  for i, b in enumerate(G_2.boundary_cycles):
    b' ← transform_boundary_cycle(f_2, b)
    i' ← index of b' in G.boundary_cycles
    τ'[i'] ← i
  return τ'
```

By assumption, $\bar{\nu}_1$ numbers the boundary cycles on G_1 in the order they are returned by running function *compute_boundary_cycles* on $\boldsymbol{G}_1$, so $\sigma_1^{(j)}$ is the permutation corresponding to the j-th element in $\boldsymbol{p}_1$.*markings*.

The map ϕ_1 is easy to compute: again, given that both $\bar{\nu}$ and $\bar{\nu}_1$ number the boundary cycles of G and G_1 in the order they are returned by *compute_boundary_cycles*, the auxiliary function *compute_phi1_inv* incrementally builds the result by looping over $\boldsymbol{G}_1$.*boundary_cycles*, contracting the target edge, and mapping the corresponding indices.

Computation of the map ϕ_2 depends on the isomorphism f_2; however, two different choices for f_2 will not change the outcome of the algorithm: in the final loop at lines 12–16, only the sign of f_2 is used, and the sign is constant across all isomorphisms having the same source and target fatgraphs (iff they are both orientable). Computation of ϕ_2^{-1} (in the auxiliary function *compute_phi2_inv*) is done in the same way as the computation of ϕ_1^{-1}, except we transform b to b' by means of *transform_boundary_cycle*$(f_2, -)$, i.e., $B(f_2)$.

Finally, for every marking $\sigma_1^{(j)}$ in $\boldsymbol{p}_1$.*markings* (representing $\nu_1^{(j)}$), we know by Lemma 3.36 that there is a unique index k and $a \in \operatorname{Aut} G_2$ such that: $\sigma_1^{(j)} \circ \phi_1^{-1} \circ \phi_2^{-1}$ is the k-th item $\sigma_2^{(k)}$ in $\boldsymbol{p}_2$.*markings* (representing $\nu_2^{(k)}$), and such that the following chain:

$$G_1 \xrightarrow[f_1]{} G \xrightarrow[f_2]{\simeq} G_2 \xrightarrow[a]{\simeq} G_2$$

extends to a marked fatgraph morphism:

$$(G_1, \nu_1^{(j)}) \xrightarrow[\hat{f}_1]{} (G, \nu) \xrightarrow[\hat{f}_2]{\simeq} (G_2, \nu_2) \xrightarrow[\hat{a}]{\simeq} (G_2, \nu_2^{(k)}).$$

The sign s is then obtained by comparing the orientation ω_2 of $(G_2, \nu_2^{(k)})$ with the push-forward orientation $(a \circ f_2 \circ f_1)_* \omega_1$, where ω_1 is the orientation on $(G_1, \nu_1^{(j)})$, and multiplying by the alternating sign from the homology differential. There are four components that make up s:

- » the sign given by the contraction f_1: this is $+1$ by definition since the "child" fatgraph G inherits the orientation from the "parent" fatgraph G_1;
- » the sign given by the isomorphism f_2: this is obtained by comparing $(f_2)_* \omega$ with ω_2, which is implemented for a generic isomorphism by the function *compare_orientations*;
- » the sign of the automorphism a of G_2 which transforms the push-forward marking into the chosen representative in the same orbit: this again can be computed by comparing $(a)_* \omega_2$ with ω_2 and only depends on the action of a on edges of G_2;
- » the alternating sign from the homology differential, which only depends on the position p of edge e within the order ω_1.

The product of the three non-trivial components is returned as the sign s. □

4.3. Matrix form of the differential D. In Algorithm 10, the function *compute_boundary_operators* computes the matrix form $\boldsymbol{D}^{(m)}$ of the differential D restricted to the linear space generated by fatgraphs with m edges.

Input to the function are the number m and the list of graphs, divided by number of fatgraph edges: *graphs*$[m]$ is the list of fatgraphs with m edges.

Algorithm 10 Compute the boundary operator matrix, block by block.

```
def compute_boundary_operator(m, graphs):
  N_1 ← number of graphs with m edges
  N_2 ← number of graphs with m − 1 edges
  D^(m) ← N_1 × N_2 matrix, initially null
  j_0 ← 0
  for G_1 in graphs[m]:
    p_1 ← MarkedFatgraphPool(G_1)
    k_0 ← 0
    for G_2 in graphs[m−1]:
      p_2 ← MarkedFatgraphPool(G_2)
      for e in G_1.edges:
        if e is a loop:
          continue with next e
        for (j, k, s) in compute_block(p_1, e, p_2):
          add s to entry D^(m)[k + k_0, j + j_0]
      increment k_0 by the number of inequivalent markings in p_2
    increment j_0 by the number of inequivalent markings in p_1
  return D^(m)
```

The output matrix $\boldsymbol{D}^{(m)}$ is constructed incrementally: it starts with all entries set to 0, and is then populated blockwise. Indeed, for every pair of *MarkedFatgraphPool* objects $\boldsymbol{p}_1$ (from a graph with m edges) and $\boldsymbol{p}_2$ (with $m-1$ edges), and every non-loop edge $\boldsymbol{e}$, the rectangular matrix block whose upper-left corner is at indices j_0, k_0 is summed the block resulting from *compute_block*($\boldsymbol{p}1$, $\boldsymbol{e}$, $\boldsymbol{p}_2$).

5. Experimental results

An implementation[12] of the algorithms presented in this paper has been actually used to compute the Betti numbers of all $\mathcal{M}_{g,n}$ with $2g+n \leqslant 6$. The results are summarized in Table 3.1. All the Betti numbers were already known in the literature; Section 2 in the "Introduction" chapter provides references. In all these cases, published results agree with the values in Table 3.1.

An internal verification step in the code computes the classical and virtual Euler characteristics of the fatgraph complex; the values computed by the Python program match those published in [28, 8, 7], where they are derived by theoretical means.

Along with the computation, the entire family of fatgraphs $\mathcal{R}_{g,n}$ (with $2g+n \leqslant 6$) has been computed, and for each fatgraph the isomorphism group is known. The full list of fatgraphs and their isomorphisms is too long to print here, but the data is publicly available at http://fatghol.googlecode.com/download/list. Tables 3.2 and 3.3 provide a numerical summary of the results.

[12] Written in the Python programming language (see [12, 65]). Code is publicly available at http://code.google.com/p/fatghol.

	b_0	b_1	b_2	b_3	b_4	b_5	b_6	b_7	b_8	b_9	b_{10}	b_{11}	b_{12}
$\mathcal{M}_{0,3}$	1												
$\mathcal{M}_{0,4}$	1	2											
$\mathcal{M}_{0,5}$	1	5	6										
$\mathcal{M}_{0,6}$	1	9	26	24									
$\mathcal{M}_{1,1}$	1												
$\mathcal{M}_{1,2}$	1												
$\mathcal{M}_{1,3}$	1			1									
$\mathcal{M}_{1,4}$	1			4	3								
$\mathcal{M}_{2,1}$	1		1										
$\mathcal{M}_{2,2}$	1		2			1							

TABLE 3.1. Betti numbers of $\mathcal{M}_{g,n}$ for $2g+n \leqslant 6$. For readability, null values have been omitted and the corresponding entry left blank.

No. of edges:	*12*	*11*	*10*	*9*	*8*	*7*	*6*	*5*	*4*	*3*	*2*	Total
$g=0$, $n=3$										2	1	3
$g=0$, $n=4$							6	6	7	6		25
$g=0$, $n=5$				26	26	72	103	65	21			313
$g=0$, $n=6$	191	191	866	1813	1959	1227	418	76				6741
$g=1$, $n=1$										1	1	2
$g=1$, $n=2$							5	5	8	8		26
$g=1$, $n=3$				46	46	162	256	198	72			780
$g=1$, $n=4$	669	669	3442	7850	9568	6752	2696	562				32208
$g=2$, $n=1$				9	9	29	52	45	21			165
$g=2$, $n=2$	368	368	2005	4931	6543	5094	2279	546				22134

TABLE 3.2. Number of distinct abstract fatgraphs with the given genus g and number of boundary cycles n. For readability, null values have been omitted and the corresponding entry left blank.

5.1. Performance. Table 3.4 gives a summary of the running times obtained on the `idhydra.uzh.ch` cluster at the University of Zurich; Figure 3.11 provides a graphical representation of the same data. The computational demands of the code are such that the homology of $\mathcal{M}_{g,n}$ can actually be computed on desktop-class hardware for $2g + n < 6$.

The scatter plot in Figure 3.11 shows that the time spent in computation of the $\boldsymbol{D}^{(m)}$ matrix ranks done in Stage III can become the dominant contribution to the total running time as the number of fatgraphs increases. This highlights a limitation of the program: the large number of fatgraphs in the Kontsevich complex might turn out to be a challenge for today's sparse linear algebra software.

However, the set of fatgraphs for a given (g, n) pair has to be generated prior to computing the matrices $\boldsymbol{D}^{(m)}$: a very large set of graphs can exhaust the computer's memory long before computation time becomes a blocking issue.

6. Application to other fatgraph complexes

In [23], V. Godin defined a "bordered fatgraph complex", which computes the integral homology of the moduli spaces of Riemann surfaces with boundaries. Godin's

No. of edges	$\mathcal{M}_{0,3}$	$\mathcal{M}_{0,4}$	$\mathcal{M}_{0,5}$	$\mathcal{M}_{0,6}$	$\mathcal{M}_{1,1}$	$\mathcal{M}_{1,2}$	$\mathcal{M}_{1,3}$	$\mathcal{M}_{1,4}$	$\mathcal{M}_{2,1}$	$\mathcal{M}_{2,2}$
12				122880				14944		713
11				616320				81504		3983
10				1274688				185760		9681
9			2240	1359840			236	227564	9	12927
8			8160	862290			918	160128	28	10077
7			11280	294480			1440	63756	43	4519
6		64	7260	49800		9	1112	13000	39	1057
5		144	2112	3024		15	408	1008	20	97
4		99	210			10	54		3	
3	4	20			1	3				
2	3				1					
Total	7	327	31262	4583322	2	37	4168	747664	142	43054

TABLE 3.3. Number of distinct orientable marked fatgraphs in the Penner-Kontsevich complex of each of the indicated $\mathcal{M}_{g,n}$ spaces. For readability, null values have been omitted and the corresponding entry left blank.

Time (s):	Stage I	Stage II	Stage III	Total
$\mathcal{M}_{0,3}$	$<$ 4ms	$<$ 4ms	0.03	0.12
$\mathcal{M}_{0,4}$	0.05	0.09	$<$ 4ms	0.29
$\mathcal{M}_{0,5}$	4.78	21.91	1.85	29.43
$\mathcal{M}_{0,6}$	2542.56	16011.70	179157.39	233007.06
$\mathcal{M}_{1,1}$	$<$ 4ms	$<$ 4ms	0.010	0.128
$\mathcal{M}_{1,2}$	0.05	0.08	$<$ 4ms	0.27
$\mathcal{M}_{1,3}$	40.56	136.88	$<$ 4ms	174.75
$\mathcal{M}_{1,4}$	82486.51	336633.75	4872.69	424615.85
$\mathcal{M}_{2,1}$	2.39	4.76	$<$ 4ms	7.39
$\mathcal{M}_{2,2}$	43402.18	181091.11	5.57	224694.61

TABLE 3.4. Total CPU time (seconds) used by the Betti numbers computation for the indicated $\mathcal{M}_{g,n}$ spaces. The C++ library LINBOX [**43**, **14**] was used for the rank computations in Stage III. Running time was sampled on the `idhydra.uzh.ch` computer of the University of Zurich, equipped with 480GB of RAM and 48 Intel Xeon CPU cores model X7542 running at 2.67GHz; Python version 2.6.0 installed on the SUSE Linux Enterprise Server 11 64-bits operating system was used to execute the program. The system timer has a resolution of 4ms. The "Total" column does not just report the sum of the three stages, but also accounts for the time the program spent in I/O and memory management.

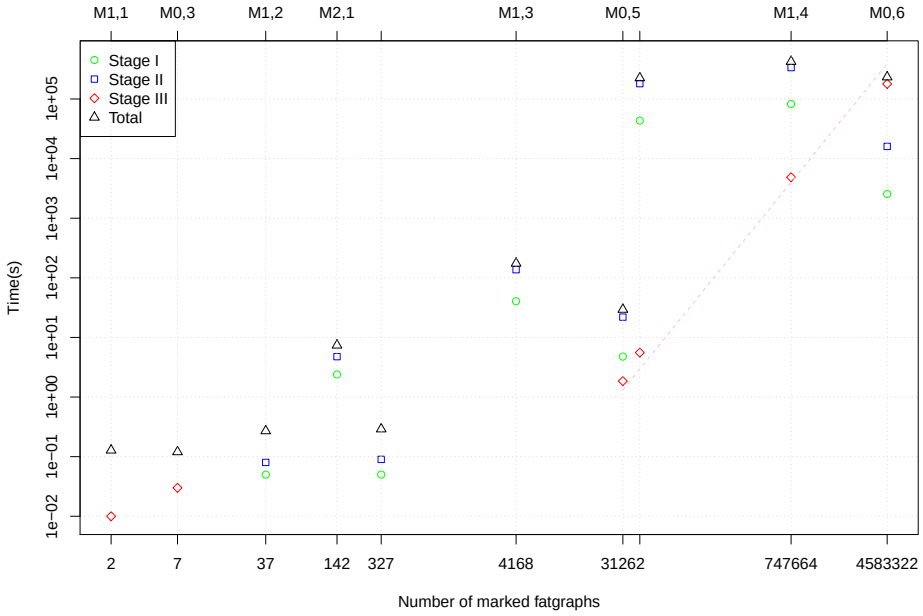


FIGURE 3.11. Scatter plot of the data in Table 3.4. Both axes use log-scale. Note how Stage III (computation of the boundary operators rank) becomes the dominant task as the number of marked fatgraphs increases.

fatgraphs extend the abstract fatgraph by requiring that a *leaf* (i.e., a univalent vertex), and only one, is present in each boundary cycle. The bordered fatgraph complex is then constructed exactly as the fatgraph complex presented here, with the proviso that an edge ending in a univalent vertex is never contracted: hence, the differential D is given by the sum of contraction of non-loop non-leaf edges.

The algorithms of this paper can easily be adapted to compute the homology of Godin's bordered fatgraph complex: after generating the family of marked fatgraphs of a given (g, n) pair, we decorate each marked fatgraph with leaves; compute the matrix form of the differential D and then reduce it to Smith normal form to reckon the *integral* homology modules of the moduli space of bordered surfaces.

There is no need for checking duplicates in the set of bordered fatgraphs so generated,[13] therefore the decoration step can be implemented efficiently. A shortcut can also be taken in computing the matrix D: since leaf edges are never contracted, the differential on bordered fatgraphs can be deduced easily from the differential on marked fatgraphs. However, the number of bordered fatgraphs is much larger than the number of marked fatgraphs;[14] this means that the final linear algebra

[13] If two "bordered fatgraphs" were isomorphic, they would remain such if we remove the leaves and the edge supporting them, which would give us isomorphic marked fatgraphs

[14] A leaf may be regarded as a choice of an edge or a vertex along a boundary cycle; if there are p_i vertices (counted with multiplicities) and q_i edges along the i-th boundary cycle, then the number of ways we could possibly add leaves to a marked fatgraph G is $r_1 r_2 \cdots r_n$, with $r_i = p_i + q_i$ so that:

$$r_1 + r_2 + \cdots + r_n = \sum_i p_i + \sum_i q_i = \sum_{v \in V(G)} z_v + 2m = 4m,$$

where m is the total number of edges and z_v is the valence of vertex v.

computations require even more computational resources than they do for $\mathcal{M}_{g,n}$ computations.

7. Future development directions

There are a number of directions in which the current algorithms and code could be improved.

As already noted, the generation algorithms produce quite a number of duplicates, that have to be removed using a quadratic-complexity procedure. A variant of the "isomorph-free generation" algorithm of McKay [47] could replace the naive *MgnTrivalentGraphs* code; the question of which algorithm would be faster has probably to be sorted out empirically, the critical performance factor being the number of times the "isomorphism" test is invoked.

Another approach would be to turn the generation procedure "upside down": instead of starting with trivalent graphs and contracting edges, one could start with (g, n)-fatgraphs with one vertex and expand those until the whole set of fatgraphs is generated. This would have the advantage that the chromatic fatgraph polynomial of Bollobás and Riordan [9] is available as an invariant to speed up the isomorphism procedure. On the other hand, the number of fatgraphs generated this way seems consistently larger than the number of fatgraphs generated with the procedure adopted here (see Section 2 in Appendix B).

So far, the major obstacle to applying the algorithms of this paper to a wider range of moduli spaces has been the large number of fatgraphs involved: it affects both the total run time and memory consumption of the code. Most algorithms described here lend themselves naturally to parallelization, so it would be possible to rewrite the program to exploit several processors and distributed memory, which could solve both issues. However, the number of generated fatgraphs grows super-exponentially in the asymptotic limit [4, 5], so any implementation of the algorithms outlined here will soon hit the limit of any present-day computing device. The question remains open, whether more significant result could be obtained before hitting the limits of today's computers.

CHAPTER 4

A novel parallel algorithm for exact Gaussian Elimination of general sparse matrices

The algorithm presented in Chapter 3 reduces computation of the Betti number of moduli spaces $\mathcal{M}_{g,n}$ to reckoning the rank (over $\mathbb{Q}$) of some large sparse matrix with integer entries. An effective method for computing this rank is given by Gaussian Elimination.

Here we describe a new algorithm for Gaussian Elimination (nicknamed "Rheinfall"); it was specifically developed for computing the rank of fatgraph homology matrices, but, being a variant of the most general Gaussian Elimination procedure, has wider applicability.

The "Rheinfall" algorithm is, in essence, a rearrangement of the operations in the usual Gaussian Elimination with Partial Pivoting (GEPP) procedure; in contrast to GEPP, the "Rheinfall" algorithm presented here naturally allows a parallel formulation both in the message-passing/distributed-memory paradigm and in the threaded/shared-memory one. However, the "Rheinfall" algorithm degrades gracefully to sequential execution when run on a single compute node, and experiments have shown that it can yield better performance than GEPP on certain classes of matrices.

A sample implementation of the code is available at http://rheinfall.googlecode.com; examples and performance data given in the text have been measured using that program for computing the rank of integer matrices (using machine-size integers as the entry type).

Any Gaussian Elimination algorithm can be applied equally well column- or row-wise; here we take the row-oriented approach.

This chapter features a large number of figures and tables, that are placed at the end of the chapter rather than being interspersed through the text.

1. Description of the algorithm

Let $A = (a_{ij} | i = 0, \ldots, n-1; j = 0, \ldots, m-1)$ be a $n \times m$ matrix with entries in a "sufficiently good"[1] ring $\Bbbk$.

Definition 4.1. Given a matrix A, define $z_i := \min\{j | a_{ij} \neq 0\}$, i.e., z_i is the column index of the first non-zero entry in the i-th row of A; for a null row, define $z_i := m$. We say that the i-th row of A *starts* at column z_i.

The matrix A is in *block echelon form* iff $z_i \geqslant z_{i-1}$ for all $i = 1, \ldots, n-1$.

The matrix A is in *row echelon form* iff, for all i, either $z_i > z_{i-1}$ or $z_{i-1} = z_i = m$.

[1] By "sufficiently good" we mean that it has enough properties to meaningfully define a Gaussian Elimination procedure: for instance, a field or a Bézout domain.

Every matrix can be put into *block* echelon form by a permutation of the rows. We shall show that the code in Algorithm 11 can be used to transform any given matrix into *row* echelon form.

The "Rheinfall" algorithm is most easily described using a Bulk Synchronous Parallel (BSP) model [64]; we show in Sections 1.1 and 1.2 how it can be adapted to popular implementation paradigms.

For reducing the $n \times m$ matrix A to row echelon form, create m Processing Units $P[0], \ldots, P[m-1]$, one for each matrix column: Processing Unit $P[c]$ handles rows starting at column c.

Each Processing Unit $P[c]$ maintains some internal state variables:

u_c Holds either a matrix row, or (initially) the special value NIL. By construction, $u_c[c] \neq 0$ and $u_c[j] = 0$ for all $j < c$.

Q_c A collection of rows on which to perform elimination. An incoming ROW message (with payload a matrix row r) triggers the addition of r to Q_c; likewise, when r is modified by elimination, it is removed from Q_c. No assumption is done on the data structure underlying Q_c, other than it allows addition and removal of rows; in particular, Q_c need not preserve the order in which rows have been added to it.

$output[c]$ The contribution of this PU to the global result: its actual form varies with the specifics of what is being computed by the algorithm. For reducing a matrix to row echelon form, *output* is a matrix row; other variants are discussed in Sections 2.1 and 2.2.

During each BSP superstep, Processing Unit $P[c]$ orderly performs the tasks below, independently from other Processing Units (PUs). PUs communicate only by exchanging ROW messages: the content of a ROW message is a matrix row r.

(1) Chooses a pivot row u_c among the rows available in block Q_c.
(2) Performs elimination on all the rows in Q_c using the chosen pivot row.
(3) Sends the modified rows to other PUs: if row r' (gotten by elimination of row r with the pivot row u_c) starts at column c', send it to PU $P[c']$.

When all processing is done, the *output* of each $P[c]$ is made available to a "master" process, which assembles the global result.

Example 4.2. A worked out example might clarify how the Rheinfall algorithm performs Gaussian Elimination.

(a) Assume we are given the following input matrix (for readability, zeroes have been omitted and the corresponding entry left blank):

$$A = \begin{pmatrix} 1 & & & 2 & 1 \\ & & & 1 & \\ 2 & & 2 & & \\ & 3 & 1 & & 1 \\ & 1 & & 1 & 5 \end{pmatrix}$$

(b) The matrix is permuted so that All rows having the first nonzero in the same column are arranged into a block of consecutive indices:

$$R_0 A = \begin{pmatrix} 1 & & & 2 & 1 \\ 2 & & 2 & & \\ & 3 & 1 & & 1 \\ & 1 & & 1 & 5 \\ & & & 1 & \end{pmatrix} \tag{S0}$$

Thus, rows the first two rows, r_1 and r_2, are sent to PU $P[1]$, then r_3 and r_4 are sent to $P[2]$, and row r_5 is assigned to $P[4]$.

(c) PUs choose a pivot in each block (indicated with italic font in the following):

$$R_0 A = \begin{pmatrix} \mathit{1} & & & 2 & 1 \\ 2 & & 2 & & \\ & 3 & 1 & & 1 \\ & \mathit{1} & & 1 & 5 \\ & & & \mathit{1} & \end{pmatrix} \tag{S1}$$

Note that pivoting is restricted to the block: there is no global search or broadcast of pivot values.

(d) Elimination is now performed by each PU independently and concurrently:

$$E_0 R_0 A = \begin{pmatrix} \mathit{1} & & & 2 & 1 \\ 0 & & 2 & -4 & -2 \\ & 0 & 1 & -3 & -14 \\ & \mathit{1} & & 1 & 5 \\ & & & \mathit{1} & \end{pmatrix} \tag{S2}$$

Note that $P[4]$ does not perform any work since it has only one row in its block.

(e) Eliminated rows are sent to the PU assigned to their new starting column and the process is performed again from step (S1).

$$R_1 E_0 R_0 A = \begin{pmatrix} \mathit{1} & & & 2 & 1 \\ & \mathit{1} & & 1 & 5 \\ & & \mathit{1} & -3 & -14 \\ & & 2 & -4 & -2 \\ & & & \mathit{1} & \end{pmatrix} \tag{S1'}$$

In the above example, note that the two eliminated rows have been sent to PU $P[3]$, that was previously empty.

Note that there is actually no separate reordering step (S0) for initially putting the input matrix A into block echelon form: rows are distributed to Processing Units as they are read. In a sense, rearrangement of the matrix is an artifact of the textual representation: "Rheinfall" chops up the matrix and operates independently on each block.

More generally, this is an *invariant* of the "Rheinfall" algorithm: by exchanging rows among PUs after every round of elimination is done, the working matrix is kept in block echelon form at all times. Indeed, let $A^{(0)} = A$. After completion of the k-th superstep we can define the working matrix $A^{(k)}$ by concatenating: the pivot row u_1 of $P[1]$ (if not NIL); the rows in Q_1, in some order (e.g., in the relative order they appeared originally in A); the pivot row (if any) of $P[2]$; then rows of Q_2 (if not empty); etc.

Lemma 4.3. *$A^{(k)}$ is in block echelon form, and is gotten from A by elementary row operations.*

PROOF. $A^{(k)}$ is in block echelon form by construction. Each row of $A^{(k)}$ is either a pivot row or a linear combination of (at most) two rows of $A^{(k-1)}$; a recursion argument proves the claim. □

The *eliminate* function at line 17 in Algorithm 11 returns a sum of suitable multiples of two rows so that the result row has a null entry in all columns $j \leqslant c$. The actual definition of *eliminate* depends on the coefficient ring of A: for Gaussian Elimination over a field, *eliminate* returns $r - (r[c]/u[c])u$; elimination over a ring involves finding α, β such that $\alpha r[c] = \beta u[c]$ and then returning $\alpha r - \beta u$. Note that $u[c] \neq 0$ by construction.

Lemma 4.4. *After at most m supersteps, each row r of the $n \times m$ input matrix A has been modified to fall in either one of these two cases: (1) it is the pivot row u_c for some Processing Unit $P[c]$, or (2) it has been completely eliminated, i.e., successive modifications have zeroed out all its entries.*

PROOF. If A is the null matrix, then all rows are null and they all fall into case *(2)*.

Otherwise, assume A has at least one nonzero row and proceed by induction on m. If $m = 1$, then one row will be chosen as pivot, and the other ones will be zeroed out by elimination after the first superstep.

If $m > 1$, let $r_1, \ldots, r_k$ be the rows of A with a nonzero entry in column 1; let $\tilde{A}$ be the submatrix formed by the rest of rows of A. In the first superstep, one of them (say, r_1) is chosen as pivot row u_1, and the other ones are transformed by the *eliminate* function into rows $r'_2, \ldots, r'_k$ having a zero in the first column. By induction, every row of the $(n-1) \times (m-1)$ matrix $\hat{A}$ formed by $r'_2, \ldots, r'_k$ and $\tilde{A}$ is modified into a pivot row or zeroed out in $m-1$ steps. But $\hat{A}^{(m-1)}$ is the lower-right submatrix of $A^{(m)}$, hence the claim holds for every row of A in m steps. □

The following statement is now immediate.

Theorem 4.5. *Algorithm 11 reduces any given $n \times m$ input matrix A to row echelon form in (at most) m supersteps.*

PROOF. By Lemma 4.4, after m steps each row has either been selected as a pivot row in some PU $P[c]$, or it has been completely zeroed out by elimination. Therefore each block of $A^{(m)}$ is comprised of just one row, i.e., $A^{(m)}$ is in row echelon form. □

1.1. Distributed-memory variant. To increase efficiency in a distributed-memory/Message Passing Interface (MPI) algorithm, we drop both the requirement that communication happens as a separate substep after all elimination has been performed, and the barrier synchronization after communication. The distributed "Rheinfall" Gaussian Elimination algorithm comprises the parallel execution of the same code by several stateful Processing Units, which continually perform elimination work and exchange messages.

So, the main problem becomes how to detect when all PUs have finished their work. As in the BSP model, we assume that every Processing Unit can send messages to every other PU. We extend the communication protocol by introducing END messages: END messages carry no payload and just signal a PU to finalize computations and then stop execution. The Gaussian Elimination procedure is complete when the END token leaves the last Processing Unit $P[m]$.

In order to guarantee that the END message is the last message that a running PU can receive, we make two assumptions on the message exchange system:

(1) that messages sent from one PU to another arrive in the same order they were sent, and
(2) that it is possible for a PU to wait until all the messages it has sent have been delivered.

Both conditions are satisfied by MPI-compliant message passing systems.

A "master" process starts the $P[1]$, ..., $P[m]$ Processing Units; feeds the initial data to them; and collects the results at the end of processing, but is not otherwise involved in the computation, and remains inactive while the PUs are running.

Theorem 4.6. *Algorithm 11 reduces any given input matrix A to row echelon form in finite time.*

PROOF. Lemma 4.4, which is the key ingredient in the proof of Theorem 4.5, does not depend on the BSP assumptions. Therefore it still holds if we replace "in at most m supersteps" with "in finite time" in the claim.

Hence, to prove the theorem we only need to show that the *Master* procedure in Algorithm 11 terminates in finite time, i.e., that each PU receives an END message in finite time.

This can easily be proved, again by induction on m: when $m = 1$, the Processing Unit $P[1]$ receives the END message from the *Master* process immediately after startup. When $m > 1$, PU $P[m]$ receives the END message in a finite time τ_0 by induction hypothesis. There are only a finite number of rows in block Q_m and no more rows can be delivered to it, since all PUs $P[c]$ with $c < m$ are now in DONE state. Thus, $P[m]$ finishes its processing in a finite time τ_1 so the END token is sent back to the *Master* in finite time $\tau_0 + \tau_1$. □

1.2. Shared-memory variant. In the shared-memory setting, Processing Units are not realized as continuously-running agents, rather as "tasks" that are scheduled on-demand when there is elimination work to be performed on a block of columns.

We assume that the run-time system supports the following features:

» It is possible to atomically set the value of a variable.
» A *mutex* lock construct is available to avoid concurrent access to the same variable by concurrently-executing threads.
» A *compare-and-swap* (CAS) operation is available.[2]
» It is possible to spawn function invocations as asynchronous tasks. A task scheduler distributes tasks to the available processing resources and will *eventually* run any scheduled task (provided each task terminates in a finite amount of time); correctness of the "Rheinfall" algorithm is not dependent on any order of execution of the scheduled tasks, i.e., the task queue needs not be a FIFO.
» It is possible for an execution thread to wait until all tasks have been executed.[3]

[2]Given three operands a, x, y, CAS atomically compares the value of memory location a with the value x and, if they are equal, sets a to the new value y. Otherwise, the value stored at a is unchanged. In any case, the value stored a prior to the CAS invocation is returned as result of the operation.

[3]This feature can easily be implemented on top of the others, by atomically increasing a counter each time a task is spawned, and atomically decreasing it each time a task finishes its execution. The waiting task is woken up when the counter reaches 0.

Algorithm 11 Reduce a matrix to row echelon form by Gaussian Elimination (distributed-memory version). *Top:* Algorithm run by processing unit $P[c]$. *Bottom:* Sketch of the "master" procedure. Input to the algorithm is an $n \times m$ matrix A, represented as a list of rows r_i. Row and column indices are 1-based.

```
1  def ProcessingUnit(c): [where c is a column index]
2    u_c ← NIL
3    Q_c ← empty list
4    output[c] ← NIL
5    state ← RUNNING
6    while state is RUNNING:
7      wait for messages to arrive
8      append ROW messages to Q_c
9      select best pivot row t from Q_c
10     if u_c is NIL:
11       u_c ← t
12       remove t from Q_c
13     else:
14       if t has better pivoting features than u_c:
15         exchange u_c and t
16     for each row r in Q_c:
17         r' ← eliminate(r,u_c)
18         c' ← first nonzero column of r'
19         send r' to P[c']
20         delete r from Q_c
21     if received message END:
22       wait for all sent messages to arrive
23       output[c] ← u_c
24       if c + 1 < m:
25         send END to P[c + 1]
26       else:
27         send END to Master
28       state ← DONE -- break out of the loop
29   return output[c]
```

```
1  def Master(A): [where A is an n × m matrix]
2    start a PU P[c] for each column c of A
3    for i in {1, ..., n}:
4      c ← first nonzero column of r_i
5      send r_i to P[c]
6    send END message to P[1]
7    wait until P[m] sends out an END message
8    result ← collect output results from all PUs
9    return result
```

Algorithm 12 Reduce a matrix to row echelon form by Gaussian Elimination (shared-memory version). Input to the algorithm is an $n \times m$ matrix A, represented as a list of rows r_i. Row and column indices are 1-based.

```
1  def Master(A): [where A is an n × m matrix]
2    for c in {1, ..., m}:
3      InitProcessingUnit(c)
4    for i in {1, ..., n}:
5      c ← first nonzero column of r_i
6      SendRow(r_i, c)
7    wait until all tasks are done
8    result ← collect output[c] from all PUs
9    return result

1  def SendRow(r, c): [where r is a matrix row and c is a column]
2    acquire lock L_c
3    add r to Q_c
4    release lock L_c
5    status ← CAS(spawned[c], FALSE, TRUE)
6    if status is FALSE: -- compare with old value of spawned[c]
7      spawn task ProcessingUnitTask(c)

1  def InitProcessingUnit(c): [where c is a column index]
2    u_c ← NIL
3    Q_c ← empty list
4    output[c] ← NIL
5    spawned[c] ← FALSE

1  def ProcessingUnitTask(c): [where c is a column index]
2    B ← empty list
3    acquire lock L_c
4    swap contents of B and Q_c
5    release lock L_c
6    select best pivot row t from B
7    if u_c is NIL:
8      u_c ← t
9    else:
10     if t has better pivoting features than u:
11       exchange u and t
12   for each row r in B:
13       r' ← eliminate(r,u)
14       c' ← first nonzero column of r'
15       SendRow(r', c')
16   output ← u
17   if Q_c is not empty:
18     -- re-schedule self --
19     spawn task ProcessingUnitTask(c)
20   else:
21     atomically set spawned[c] to FALSE
22   return output
```

All these features are easily available in modern systems; the source code freely available at http://rheinfall.googlecode.com/ provides a sample C++ implementation on top of the Intel "Threading Building Blocks" library [35].

The definition of a Processing Unit $P[c]$ is augmented with two more internal variables:

L_c A mutex lock object that protects modification of the block of rows Q_c: a section of code must acquire the lock prior to any modification. If the lock is currently being held by another task, the task wishing to acquire the lock shall wait until the lock is released.

$spawned[c]$ A boolean flag that is TRUE when a task to run *ProcessingUnitTask*(c) has already been spawned; it serves as a guard for not spawning two concurrent tasks operating as the same PU. Write accesses to $spawned[c]$ must exclude concurrent reads.

The "Rheinfall" algorithm is applied to a matrix by passing it as an argument to the *Master* procedure of Algorithm 12. The *Master* process kick-starts PU $P[c]$ for all $c = 1, \ldots, m$ such that there is at least one row starting at column c; then it waits for all tasks to be finished before collecting the results.

Theorem 4.7. *Algorithm 12 reduces any given input matrix A to row echelon form in finite time.*

PROOF. As in the proof of Theorem 4.6, the validity of Lemmas 4.3 and 4.4 does not depend on the BSP model: we just need to prove that each matrix row undergoes the elimination steps in finite time.

A *ProcessingUnitTask* is scheduled for each non-empty block in A's block echelon form (lines 4–6 in *Master* of Algorithm 12). Since, by assumption, the task scheduler will eventually execute any task, every row is either selected as pivot row or modified by elimination and sent to a higher-column PU; a recursion argument proves the thesis.

So we are left with proving that every instance of *ProcessingUnitTask*(c) completes in finite time. The only operations that could block *ProcessingUnitTask* indefinitely are the locking operations on lines 3, 15 (indirectly because of line 2 in *SendRow*). If lock contention occurs, we prioritize contenders by stipulating that the task associated with the lowest column index c wins.[4] Since the number of rows processed by a PU is finite, this guarantees that eventually every PU will get a chance to acquire the lock and complete its tasks. □

1.3. Sequential execution. The "Rheinfall" algorithm can also be executed sequentially. We shall not discuss this at length, as sequential execution is a degenerate case of the shared-memory algorithm, where only one task is running at a time, and the tasks scheduler strictly prioritizes tasks by ascending column index. Of course, there is no need for locking in the sequential execution model, so the locking operations can be left out.

[4]This requirement can be relaxed: the sample implementation uses TBB's `queuing_mutex` class [36], which guarantees that lock contenders are queued and will eventually be granted the lock.

2. Applications

2.1. Computation of matrix rank. The Gaussian Elimination algorithm can be easily adapted to compute the rank of a general (unsymmetric) sparse matrix: one just needs to count the number of non-null rows of the row echelon form.

The *output*[c] variable is modified to hold an integer number: the result shall be 1 if at least one row has been assigned to PU $P[c]$ ($u \neq$ NIL) and 0 otherwise.

Procedure *Master* performs a sum-reduce when collecting results: replace line 8 in Algorithm 11 and line 8 in Algorithm 12 with *result* $\leftarrow$ sum-reduce of *output* from $P[c]$, for $c = 1, \ldots, m$.

2.2. LUP **factorization.** We shall outline how Algorithm 11 can be modified to produce a variant of the familiar LUP factorization. For the rest of this section we assume that A has coefficients in a field and is square and full-rank.

It is useful to recast the Rheinfall algorithm in matrix multiplication language, to highlight the small differences with the usual LU factorization by Gaussian Elimination. Let Π_0 be the permutation matrix that reorders rows of A so that $\Pi_0 A$ is in block echelon form; this is where Rheinfall's PUs start their work. We can write the k-th elimination step as multiplication by a matrix E_k (which is itself a product of elementary row operations matrices), and the ensuing communication step as multiplication by a permutation matrix Π_{k+1} which rearranges the rows again into block echelon form (with the proviso that the u row to be used for elimination of other rows in the block comes first). In other words, after step k the matrix A has been transformed to $A^{(k)} = E_k \Pi_{k-1} \cdots E_0 \Pi_0 A$.

Theorem 4.8. *Given a square full-rank matrix A, the Rheinfall algorithm outputs a factorization $\Pi A = LU$, where:*

- $U = E_{n-1}\Pi_{n-1} \cdots E_0 \Pi_0 A$ *is upper triangular;*
- $\Pi = \Pi_{n-1} \cdots \Pi_0$ *is a permutation matrix;*
- $L = \Pi_{n-1} \cdots \Pi_1 \cdot E_0^{-1} \Pi_1^{-1} E_1^{-1} \cdots \Pi_{n-1}^{-1} E_{n-1}^{-1}$ *is lower unitriangular.*

PROOF. Clearly we have $\Pi A = LU$ with Π, L, U as given in the statement of the theorem. By Theorem 4.5, we know that U is upper triangular, so we only need to show that L is lower unit triangular.

To prove that L is lower unitriangular, consider the sequence of matrices recursively defined by $L_0 = E_0^{-1}$ and $L_{k+1} = \Pi_{k+1} L_k \Pi_{k+1}^{-1} E_{k+1}^{-1}$: we shall show by induction on k that L_k has unit diagonal and its leading $k \times k$ submatrix is lower unitriangular. The proof of this statement relies on two observations:

(1) At step k, Processing Unit $P[k]$ has finished processing all its rows and no new ones can arrive, so Π_k fixes rows $\{0, \ldots, k-1\}$, and the leading $(k+1) \times (k+1)$ submatrix of E_k is the identity matrix.
(2) The row used as pivot is the lowest-numbered one in a block, so E_k (hence E_k^{-1}) is lower unitriangular.

The base case $k = 0$ is an immediate consequence of these two assertions.

When $k > 0$, by induction we have that L_k has unit diagonal and a leading $k \times k$ lower unitriangular submatrix. Let $\pi_{k+1} \in \mathfrak{S}_n$ be the permutation corresponding to

the permutation matrix Π_{k+1}: we then have $(\Pi_{k+1}L_k\Pi_{k+1}^{-1})_{i,j} = (L_k)_{\pi_{k+1}(i),\pi_{k+1}(j)}$. In the course of elimination, rows only move towards higher-numbered PUs, so row $(k+1)$ can only have been modified by linear combination with lower-numbered rows; hence, $(L_k)_{kl} = 0$ for $l > k$, and $(L_k)_{lk} = 0$ for $l < k$. Thus the leading $(k+1) \times (k+1)$ submatrix of $\Pi_{k+1}L_k\Pi_{k+1}^{-1}$ is lower unitriangular, and the same holds for $L_{k+1} = \Pi_{k+1}L_k\Pi_{k+1}^{-1} \cdot E_{k+1}^{-1}$. □

The *LUP* algorithm variant works by exchanging triplets (r, h, s) among PUs; every PU stores one such triple (u, i, l), using u as pivot row. Each processing unit $P[c]$ receives a triple (r, h, s) and sends out (r', h, s'), where:

- The r rows are initially the rows of $\Pi_0 A$; they are modified by successive elimination steps as in Algorithm 11: $r' = r - \alpha u$ with $\alpha = r[c]/u[c]$.
- h is the row index at which r originally appeared in $\Pi_0 A$; it is never modified.
- The s rows start out as rows of the identity matrix: $s = e_h$ initially. Each time an elimination step is performed on r, the corresponding operation is performed on the s row: $s' = s + \alpha l$.

When the last PU has terminated its job, the *Master* procedure collects triplets (u_c, i_c, l_c) from PUs and constructs:

- the upper triangular matrix $U = (u_c)_{c=1,\dots,n}$;
- a permutation π of the indices, mapping the initial row index i_c into the final index c (this corresponds to the Π permutation matrix);
- the lower triangular matrix L by assembling the rows l_c after having permuted columns according to π.

3. Algorithm characteristics

The "Rheinfall" parallel algorithm operates closely like the sequential Gaussian Elimination, its strength being chiefly that each processing unit can independently perform elimination on a subset of the rows. However, the processing units are (logically) completely independent processes, and it is thus quite difficult to reason about their their collective performance and provide a detailed analysis of the algorithmic complexity.

We take therefore and experimental and statistical approach. We begin by reviewing some simple cases where the complexity can be worked out in detail.

3.1. Three examples of predictable performance. *(1)* Worst-case performance is attained on the $n \times (n+1)$ matrix $\Lambda^{(n)}$ defined by:

$$\Lambda_{ij}^{(n)} = \begin{cases} 1 & \text{if } j = 0 \text{ or } j = i+1, \\ 0 & \text{otherwise.} \end{cases}$$

It is clear that $\Lambda^{(n)}$ is already in block echelon form, comprising a single block which is sent to Processing Unit $P[0]$: elimination using the first row as pivot row produces the matrix $\Lambda^{(n-1)}$ as a result, which is then sent to $P[1]$, and so on. Thus, Rheinfall (any variant) collapses to *sequential* operation, and complete elimination of $\Lambda^{(n)}$ requires $n^2/2$ row operations and the exchange of $O(n^2)$ ROW messages. A different pivoting strategy can help prevent this collapse; see the discussion on "weight pivoting" in Section 3.2 below.

$$\left(\begin{array}{cccccc|c} 1 & 1 & 1 & 1 & 0 & 0 & r_0 \\ 3 & 1 & 1 & 0 & 0 & 0 & r_1 \\ 5 & 0 & 0 & 1 & 0 & 1 & r_2 \\ 7 & 1 & 1 & 1 & 0 & 0 & r_3 \end{array}\right)$$

FIGURE 4.1. A block of rows to provide examples of different pivoting strategies. Row r_0 would be the pivot row chosen by the "no pivoting" strategy and by threshold pivoting with $\gamma = 1/2$. Row r_1 would be chosen as pivot row by the "sparsity" pivoting strategy. Row r_2 would instead be the choice done by the "weight" pivoting strategy algorithm.

Note that only about $(1/n)$ of the entries of $\Lambda^{(n)}$ are nonzero, which proves that Rheinfall's performance does not depend on the fill percentage of the input matrix.

(2) The matrix $\bar{\Lambda}^{(n)}$ defined by $\bar{\Lambda}^{(n)}_{ij} = \Lambda^{(n)}_{i,n-j}$ (that is, we are flipping $\Lambda^{(n)}$ horizontally) is already in *row echelon* form: hence its processing by Rheinfall requires *no* row operations, and the algorithm terminates in the minimum required time. In the distributed-memory / MPI case, this could still be a significant lapse of time (the time required for the END token to travel from $P[1]$ to $P[m]$), but in the shared-memory and sequential execution case this means that a single iteration over PUs concludes the processing. This suggests that an initial reordering of the matrix by columns could speed up the elimination phase; what exact criterion to use for this rearrangement in the general case is still unclear, and could be the subject of future research.

(3) Finally, transposing $\Lambda^{(n)}$ gives a matrix $V^{(n)}$ which has a very different block echelon form, consisting of a block with *two* lines starting at column 0, and $n-1$ blocks with *one* line each. Therefore Rheinfall's processing of $V^{(n)}$ requires only $n-1$ row operations; it still collapses to sequential execution, but this time only one ROW message is sent from one PU to the next one, so the total processing time is only slightly larger than the one required by $\bar{\Lambda}^{(n)}$.

3.2. Pivoting strategies. The key observation in Rheinfall is that all rows assigned to a PU start at the same column. This implies that pivoting is restricted to the rows in a block, but also that each PU may independently choose the row it shall use for elimination.

This allows most arguments about GEPP to be ported almost verbatim to "Rheinfall". The main difference with the usual column-scope partial pivoting is that different pivot rows may be used at different times: when a new row with a better pivoting entry arrives, it replaces the old one. This can lead to a sub-optimal pivot being used for the early eliminations, before the "best" pivot row settles into place. However, as it is very difficult to predict what rows will form the block Q_c at a certain stage of the algorithm, only experiments can tell when and where this is an issue in practice.

Four different pivoting strategies have been evaluated in the sample Rheinfall implementation. Figure 4.1 provides an example of what the different strategies would select as pivot within a block of rows.

3.2.1. *No pivoting.* In this strategy, the first row to ever arrive to a PU $P[c]$ is used as the pivot row u_c. In the sample implementation, this is the lowest-numbered row that starts at column c. In the example in Figure 4.1, row r_0 would be chosen as pivot since it was the first to be inserted in the block.

3.2.2. *Sparsity pivoting.* At each round of elimination performed on the block Q_c, the row with less nonzero entries is chosen as pivot. Ties are resolved by choosing the row with the leading entry of smallest absolute value (if performing elimination over the integers; on floating-point entries the largest entry should be chosen to ensure numerical stability).

In the example in Figure 4.1, row r_1 would be chosen as pivot: it has the same number of entries as row r_2, but the leading coefficient is smaller in value.

3.2.3. *Weight pivoting.* Define the *weight* $W(r)$ of a matrix row r by $W(r) := \sum\{1/j : r[j] \neq 0\}$. At each round of elimination, choose the row with the lowest weight as the pivot row.

This is a variant of sparsity pivoting that has been developed to avoid the slowdown caused by $\Lambda^{(n)}$-like matrices (see example *(1)* in Section 3.1 above). Indeed, when performing elimination on a $\Lambda^{(n)}$ matrix with weight pivoting, then the last row $\Lambda^{(n)}_{n,*}$ will be chosen as pivot row; elimination in $P[1]$ then yields a matrix having the same nonzero pattern as $\bar{\Lambda}^{(n-1)}$ (see example *(2)* above), which is processed very efficiently.

In the example in Figure 4.1, row r_2 would be chosen as pivot; this shows that weight-based pivoting tries to select rows whose nonzero entries are more dense in the final segment of the row.

3.2.4. *Threshold pivoting.* Threshold pivoting interpolates between sparsity- or weight-based pivoting and the choice of pivot row by the norm of the entry in leading position (as usual in GEPP).

Fix $\gamma \in [0, 1]$ and let $Q_c^+ = Q_c \cup \{u_c\}$ be the block of rows worked on by Processing Unit $P[c]$ at a certain point in time (including the current pivot row u_c).

If A has integer entries, let $b = \min\{|r[c]| : r \in Q_c^+\}$ and choose as pivot the row r of minimum weight among those such that $|r[c]| \leqslant b/\gamma$.

If A has floating-point entries, then let $b = \max\{|r[c]| : r \in Q^+\}$ and choose as pivot the row r that minimizes weight among those such that $|r[c]| \geqslant \gamma \cdot b$. This guarantees that elements of L are bounded by γ^{-1}.

In the example in Figure 4.1, row r_0 would be chosen as pivot: it has the smallest leading entry and no other row is within a factor of 2 from it.

3.3. Complexity estimates. In order to assess the impact of the four pivoting strategies described in section 3.2 above, we have run the sample Rheinfall implementation on all matrices of the SIMC collection, counting the total number of arithmetic operations performed by the rank-computing algorithm for each matrix and pivoting strategy. Figure 4.2 shows a scatter plot of the total number of arithmetic operations performed by the rank-computing Rheinfall algorithm, versus the number of nonzero entries in any matrix of the SIMC collection; the complete data is available in Section 2. This computational experiment also showed that "Rheinfall" consistently fails on a certain percentage of the matrices in the SIMC collection, either because the entries get too large (arithmetic complexity overflow) or because of the growth of the fill-in (memory overflow); a more thorough theoretical assessment would be needed, but has not been carried out in the context of the present research.

Pivoting Strategy	s_{20}	s_{50}	s_{80}
No pivoting	1.1964	1.3853	2.0780
Sparsity	1.2198	1.4124	2.0780
Threshold	1.2198	1.4124	2.0780
Weight	1.2198	1.4124	2.0780

TABLE 4.1. Slopes of lines separating a certain percentage of the plot points in Figure 4.2. The value of s_P is such that a percentage P of points in the scatter plot of Figure 4.2 lies below the line with slope s_P.

An interesting pattern emerging in Figure 4.2 is that the points are arranged into a roughly triangular shape. Let N be the number of nonzero entries and K be the total number of arithmetic operations performed by the rank-computing Rheinfall algorithm. The data in Table 4.1 shows that for 80% of the matrices in the SIMC collection,

$$s_{20} \cdot \log N \leqslant \log K, \qquad \log K \leqslant s_{80} \cdot \log N.$$

Hence:

» $K \leqslant c_1 \cdot N^{2.08}$ for 80% of the SIMC matrices, and
» $K \leqslant c_2 \cdot N^{1.42}$ for 50% of the SIMC matrices,

independently of the chosen pivoting strategy; slightly sharper bounds can be given for threshold pivoting and weight-based pivoting (the exact values are printed in Table 4.1). Of course, how much these results extend to arbitrary integer-valued sparse matrices depends on how much SIMC is a representative sample of the set of "interesting" matrices.

While not a proof of the complexity characteristics of Rheinfall, this statistics shows that Rheinfall is practically advantageous over traditional GEPP for a large number of matrices.

3.4. Numerical stability. When $\gamma = 1$, threshold pivoting reduces to partial pivoting (albeit restricted to block-scope), and one can repeat the error analysis done in [24, Section 3.4.6, p. 115] almost verbatim. The main difference with the usual column-scope partial pivoting is that different pivot rows may be used at different times: when a new row with a better pivoting entry arrives, it replaces the old one. This could result in the matrix growth factor being larger than with GEPP; only numerical experiments can tell how much larger and whether this is an issue in actual practice. However, no such numerical experiments have been carried out, as this initial exploration of the Rheinfall algorithm has been driven by the needs of homology computations over $\mathbb{Q}$.

Still, the major source of instability when using the Rheinfall algorithm on matrices with floating-point entries is its sensitivity to "compare to zero": after elimination has been performed on a row, the eliminating PU must determine the new starting column (in order to forward it to the another PU). This requires scanning the initial segment of the (modified) row to determine the column where the first nonzero lies. Changes in the threshold $\epsilon > 0$ under which a floating-point number is considered zero can significantly alter the final outcome of Rheinfall processing.

4. Sequential performance

The "Rheinfall" algorithm can of course be run on just one processor: processing units execute a single `step()` pass (corresponding to lines 16–28 in Algorithm 11), one after another; this continues until the last PU has switched to DONE state.

4.1. Sample Implementation. A sample program has been written that implements matrix rank computation and LU factorization with the variants of Algorithm 11 described before. Source code is publicly available from http://code.google.com/p/rheinfall.

The sample code is written in object-oriented style, using the C++ programming language. Processing Units are implemented by a `ProcessingUnit` class, exposing a `step()` method which performs a single pass of the main loop in procedure *ProcessingUnit* (cf. lines 16–28 in Algorithm 11). A Processing Unit's "inbox" Q_c is implemented as a list of rows. When a PU starts its `step()` procedure, it performs elimination on all rows in Q_c and immediately sends the modified rows to other PUs for processing. As all PUs reside within the same OS process, communication among PUs has virtually no cost: it is implemented by simply adding a row to another PU's "inbox" Q_c.

In the sequential implementation, the main computation function consists of a loop that calls each PU's `step()` in turn, until all PUs have performed elimination. Since $P[c]$ can only receive rows from $P[c']$ if $c' < c$, one pass is sufficient to run the complete elimination procedure on any matrix.

The main program reads a file in LinBox' SMS format [15], creates a PU for each column, and dispatches rows directly to the responsible PU;[5] after that, it calls the main computation function. At the end of the computations, the total rank is computed by summing the contributions of each PU.

4.2. Integer performance. In order to get a broad picture of Rheinfall's sequential performance, the rank-computation program has been tested an all the integer matrices in the SIMC collection [15], and in particular on the $\mathcal{M}_{g,n}$ homology matrices which were the main driver for developing the "Rheinfall" algorithm. The complete data is collected in Section 3, comparing the performance of the sample Rheinfall implementation to the integer GEPP implementation provided by the free software library LINBOX [14, 43]; a summary plot is shown in Figure 4.3, showing that "Rheinfall" outperforms LINBOX in a large number of cases. Detailed results for the $\mathcal{M}_{g,n}$ matrices and a selection of those for other matrices are shown in Tables 4.2 and 4.3.

Results in Table 4.2 consistently show "Rheinfall" outperforms LINBOX, by even a couple order of magnitudes in the case of large matrices. Instead, results in Table 4.3 show great variability: the relative speed of "Rheinfall" vs LINBOX changes by orders of magnitude in one or the other direction. The performance of both algorithms varies significantly depending on the actual arrangement of nonzeroes in the matrix being processed, with no apparent correlation to simple matrix features like size, number of nonzeroes or fill percentage.

Table 4.4 shows the running time on the transposes of the test matrices. Both in LINBOX's GEPP and in "Rheinfall", the computation times for a matrix and its

[5] This implements the virtual "reordering step" needed to bring a matrix into block echelon form.

transpose could be as different as a few seconds versus several hours! However, the variability in Rheinfall is greater, and looks like it cannot be explained by additional arithmetic work alone; the examples discussed in Section 3.1 suggest that matrices showing large variability might have one or more minors of the form $\Lambda^{(k)}$. More investigation is needed to better understand how "Rheinfall" workload is determined by the matrix nonzero pattern.

4.3. Floating-point performance. In order to assess the "Rheinfall" performance in floating-point uses cases, the LU factorization program has been tested on a subset of the test matrices used in [26]. Results are shown in Table 4.5, comparing the Mflop/s attained by the "Rheinfall" sample implementation with the performance of SuperLU 4.2 on the same platform.

The most likely cause for the huge gap in performance between "Rheinfall" and SuperLU lies in the strict row-orientation of "Rheinfall": SuperLU uses block-level operations, whereas Rheinfall only operates on rows one by one. In support of this view, let us observe that the Mflop/s rate of the Rheinfall LU factorization program is higher (sometimes approximately double) than the equivalent rate of the Rheinfall rank-computation program. This can only be explained by the fact that the LU computations require two row operations per cycle in a PU's inner loop, whereas the rank computation only performs one. However, row orientation is a defining characteristic of the "Rheinfall" algorithm (as opposed to a feature of its implementation) and cannot be circumvented. Considering also the "compare to zero" issue outlined in Section 3.4, one must conclude that "Rheinfall" is generally not suited for inexact computation.

5. Parallel performance and scalability: distributed-memory

The "Rheinfall" parallel algorithm operates closely like sequential Gaussian Elimination, its strength being chiefly that each processing unit can independently perform elimination on a subset of the rows, and that communication can be completely overlapped with computation (to the extent allowed by practical implementations). However, the processing units are (logically) completely independent processes, reacting to each other's messages; it is thus quite difficult to reason about their collective performance.

5.1. Sample Implementation. The sample code presented in Section 4.1 can also be compiled with MPI [48, 49] support to run in a distributed-memory environment.

In the distributed-memory implementation, the main computation function consists of an inner loop that calls each PU's `step()` in turn, until all PUs have performed one round of elimination. Incoming messages from other MPI processes are then received and dispatched to the destination PU. After that, PUs which have transitioned to End state are removed from the list (they are all concentrated in the initial segment), and another pass is made. This outer loop repeats until there are no more PUs in Running state.

Communication is effected using the `MPI_Issend` function: each PU maintains a list of sent messages and checks at the end of an elimination cycle which ones have been delivered and can be removed. Incoming messages are only received at the end of the main inner loop, and dispatched to the appropriate PU.

Because of the way MPI messaging works, messages cannot be addressed to a specific PU: a dedicated section in the rank-computing function (executed sequentially at the end of the outer main loop) receives messages, inspects their content to find the starting column, and then enqueues each message into the appropriate PU's "inbox" Q_c.

5.2. Workload distribution. Since there is only a limited degree of parallelism available on a single computing node, an issue arises on how to map Processing Units to actual threads in the computer. The "Rheinfall" algorithm does not impose any fixed scheme for mapping PUs to execution units. A column-cyclic distribution pattern has been currently implemented, but the code is open for modification and experimentation of different schemes.

Processing units have not been implemented as separate continuously-running threads; rather, each MPI process (rank) is assigned a number of PUs and steps each of them in turn.

Let p be the number of MPI processes available, and m be the total number of columns in the input matrix A. The input matrix is divided into vertical stripes, each comprised of w adjacent columns. Stripes are assigned to MPI ranks in a cyclic fashion: MPI process k (with $0 \leqslant k < p$) hosts the k-th, $(k+p)$-th, $(k+2p)$-th, ... stripe; in other words, it owns processing units $P[w\cdot(k+a\cdot p)+b]$ where $a = 0, 1, \ldots$ and $0 \leqslant b < w$.

5.3. Experimental results. In order to assess the parallel performance and scalability of the sample "Rheinfall" implementation, the rank-computation program has been run on a small set of matrices (extracted from the larger SIMC [15] collection; see Appendix D for details). The program has been run with a varying number of MPI ranks, ranging from 16 to 256 in a geometric progression of ratio 2, and different values of the stripe width parameter w, ranging from 1 to 4096 in a geometric progression of ratio 4.

The plots in Figure 3.11 show that running time generally decreases with higher w and larger number p of MPI ranks allocated to the computation, albeit not regularly. This is particularly evident in the plot of running time versus stripe width (Figure 3.11, bottom), which shows an alternation of performance increases and decreases.

The following discussion and the examples in Section 3.1 suggest that the test matrix that perform badly have a large minor of the form $\Lambda^{(k)}$. The evidence is however not conclusive so this should be considered a working hypothesis; a more detailed investigation is needed.

The w parameter influences communication in two different ways. On the one hand, there is a lower bound $m\tau_0/w$ on the time required to pass the END message from $P[0]$ to $P[m]$, where τ_0 is the minimal time required to send an END message across the network, and we assume that communication between two PUs residing on the same compute node is instantaneous. Indeed, since the END message is always sent from one PU to the next one, then we only need to send one END message per stripe. This could explain why the running time is almost the same for large p when $w = 1$ in the good cases: the computation time decreases so much that the running time is dominated by the time taken to pass the END message along.

The plots in Figure 4.4 highlight an additional cause: when the "Rheinfall" sample implementation is run on identity $N \times N$ matrices, we know from Section 3.1 that

no elimination work is being performed, hence the total running time measures the total communication time restricted to the END messages. However, the ratio between the time spent receiving the MPI message and the time spent stepping the PUs has to be taken into account:

- for a relatively small matrix (like the $10^4 \times 10^4$ identity matrix I10000), the portion of the main loop spent in PU code is small, hence run times tend to show the erratic behavior due to network jitter (this is visible in the curves for high values of p and w);
- using a larger matrix (like the $10^7 \times 10^7$ identity matrix I10000000), the main loop still spends a significant amount of time just calling the `step()` method of each PU; hence, the run time decreases proportionally with the number of MPI ranks p and stripe width w: the larger the values of either one, the smaller the ratio of computation to communication time.

On the other hand, MPI messages are collected after every processing unit residing on a MPI rank has performed a call to its `step()` method and the needed elimination work; this leads to the risk that a single PU can slow down the entire MPI rank if it gets many elimination operations to perform. An empirical test has been conducted in the following way. The percentage of running time spent executing MPI calls has been collected using the `mpiP` tool [66]; a selection of relevant data is available in Table 4.14. The three call sites for which data is presented measure three different aspects of communication and workload balance:

- The `MPI_Recv` figures measure the time spent in actual row data communication (the sending part uses `MPI_Issend`).
- The `MPI_Iprobe` calls are all done after all PUs have performed one round of elimination: thus they measure the time a given MPI rank has to wait for data to arrive.
- The `MPI_Barrier` is only entered after all PUs residing on a given MPI rank have finished their job; it is thus a measure of workload imbalance.

Now, processing units corresponding to higher column indices naturally have more work to do, since they get the rows at the end of the elimination chain, which have accumulated fill-in. Because of the way PUs are distributed to MPI ranks, a larger w means that the last MPI rank gets more PUs of the final segment: the elimination work is thus more imbalanced. This is indeed reflected in the profile data of Table 4.14: one can see that the maximum time spent in the final `MPI_Barrier` increases with w and the number p of MPI ranks, and can even become 99% of the time for some ranks when $p = 256$ and $w = 4096$.

Finally, a larger w speeds up delivery of ROW messages from $P[c]$ to $P[c']$ iff $(c' - c)/w \equiv 0 (\mathrm{mod}\ p)$. Whether this is beneficial is highly dependent on the structure of the input matrix; some internal regularity of the input data may result on elimination work being concentrated on the same MPI rank, thus slowing down the whole program. Indeed, the large percentages of time spent in `MPI_Iprobe` for some values of p and w show that the matrix nonzero pattern plays a big role in determining computation and communication in Rheinfall. Static analysis of the entry distribution could help determine an assignment of PUs to MPI ranks that keeps the work more balanced.

6. Parallel performance and scalability: shared-memory

6.1. Sample Implementation. The sample code can also be compiled together with the Intel Threading Building Blocks (TBB) library to run on multi-core shared-memory computers.

As in the distributed-memory variant, an implementation problem to be solved is how to map Processing Units to actual threads in the computer; the solution adopted is discussed in Section 6.2 below.

In all other aspects, the shared-memory implementation closely resembles the sequential one; in particular, communication costs are negligible since passing a ROW message amounts to just moving a pointer to a row object from one `ProcessingUnit` instance to another.

6.1.1. *OpenMP.* An attempt has also been made at OpenMP-based parallelization, which has distinctly different features than TBB.

In a shared-memory OpenMP setting, the sequential main loop is naïvely modified by using an OpenMP "parallel for": each thread would call the the `step` procedure of a segment of $w = m/p$ adjacent PUs.[6] As in the sequential execution model, message delivery costs are negligible. However, w caps the number of PUs that can turn to DONE state in a single iteration: when $P[c + w - 1]$ emits the END message, $P[c + w]$'s `step` has already been run by another thread. This introduces a significant delay in the processing of the END message by $P[c + w]$.

Experimental results show indeed that the delay so introduced can be quite substantial, and that the single-node OpenMP implementation is slower than the sequential one for low values of p, and becomes only marginally faster for higher values of p.

Therefore, no further mention of OpenMP will be done in the sequel.

6.2. Workload distribution. Processing units have not been implemented as separate continuously-running threads; rather, the main computation function starts a thread pool and enqueues one task to invoke each PU's `step()` function; it then waits for termination of those tasks and the ones spawned by them.

The sample implementation offers a "coarse grained" and "fine grained" variant: the former corresponding exactly to Algorithm 12, the latter instead uses a separate task for every elimination operation.

6.3. Experimental results. To evaluate the scalability of the "Rheinfall" algorithm in the shared-memory variant, the sample implementation has been run on the same set of matrices used for testing the MPI variant. Results for the "coarse grained" variant are reported in Table 4.7 and plotted in Figure 4.7; corresponding data for the "fine grained" variant are available in Table 4.11 and graphed in Figure 4.9.

In contrast with the distributed memory approach, communication costs are negligible and there is no issue of workload distribution in the algorithm itself: the Intel TBB scheduler is responsible for managing the task queue that is created by the

[6] OpenMP offers several scheduling algorithms that partition the range in slightly different ways. Their actual performance is very close and the differences in OpenMP scheduling algorithms do not affect the following analysis of why OpenMP-based parallelization is not effective for Rheinfall.

algorithm. However, the plots in Figure 4.7 show the expected scalability behavior up to a certain number of threads (variable with the actual matrix), and then the performance stabilizes (in the good cases GL7d24 and IG5-18) or deteriorates (in the case of the $\mathcal{M}_{g,n}$ homology matrices M0,6-D8, M0,6-D9 and M0,6-D10). The outlook for the "fine grained" case is qualitatively similar, but the run times are higher and scalability issues are even more relevant. This seems to indicate that the actual computation time in the tasks is too small compared to the thread scheduling overhead by TBB.

To test this hypothesis, we run the same rank-computing program on a set of identity square matrices, with size ranging from 10^4 to 10^7; the run times are given in Tables 4.9 and 4.13, and graphed in the IV quadrant of Figures 4.7 and 4.9. By the discussion in Section 3.1, we know that "Rheinfall" performs no operations in the case of a matrix which is already in row echelon form, hence the processing time is entirely due to the thread scheduling; there is also no lock contention issue, since there is no row movement. The plots show indeed that the performance is irregular on the identity matrix, and that run time becomes larger as the number of available threads increases. This again suggests that the number of threads should be balanced with respect to the elimination work that has to be done; predicting this "break even point", however, looks like a non-trivial specific challenge and might constitute the subject of future research.

One enhancement that could help the performance goal is the introduction of a more flexible priority system in the Intel TBB library. As of this writing, TBB 4.0 only allows threads to be assigned priorities drawn from a limited discrete set (high-, normal- or low-priority tasks), whereas tasks generated by the "Rheinfall" algorithm naturally have an integer-based priority: tasks generated by rows of lower index should run before tasks generated by higher-numbered ones, in order to streamline the data flow.[7]

7. Conclusions and future work

The "Rheinfall" algorithm is basically a different way of arranging the operations of classical Gaussian Elimination, with a naturally parallel and distributed-memory formulation. It retains some important features from the sequential Gaussian Elimination; namely, it can be applied to general sparse matrices, and is independent of matrix entry type. Pivoting can be done in Rheinfall with strategies similar to those used for GEPP; however, Rheinfall is not equally suited for exact and inexact arithmetic.

Poor performance when compared to state-of-the-art algorithms and some inherent instability due to the dependency on detection of nonzero entries suggest that "Rheinfall" is not a convenient alternative for floating-point computations.

For exact arithmetic (e.g., integers), the situation is quite the opposite: up to our knowledge, "Rheinfall" could be the first practical distributed-memory Gaussian Elimination algorithm meeting the requirements of [16]. In addition, it is competitive with existing implementations also when running sequentially.

An open question, especially when comparing "Rheinfall" with other Gaussian Elimination implementations, concerns the growth of matrix entries: a more thorough analysis is needed to assess the limitations of "Rheinfall" in this respect.

[7] Priorities could as well be drawn from the real interval $[0, 1]$: tasks generated from the processing of row i are assigned priority i/n.

The distributed-memory formulation of "Rheinfall" can easily be mapped on the MPI model for parallel computations. An issue arises on how to map Rheinfall's Processing Units to actual MPI execution units; the simple column-cyclic distribution discussed in this paper was found experimentally to have poor workload balance. Since the workload distribution and the communication graph are both determined by the matrix nonzero pattern, a promising future direction could be to investigate the use of graph-based partitioning to determine the distribution of PUs to MPI ranks.

As is often the case with practical implementations, there is still room for optimizing and speeding up the code used for experiments: the current implementation focused on ease of extensibility and re-use of standard C++ library features, but this came at the expense of some performance.

At any rate, the discussion and computational experiments show that "Rheinfall" completely met its goal of being a faster rank-computation algorithm for Stage III of the graph homology computations on the moduli space of smooth pointed Riemann surfaces.

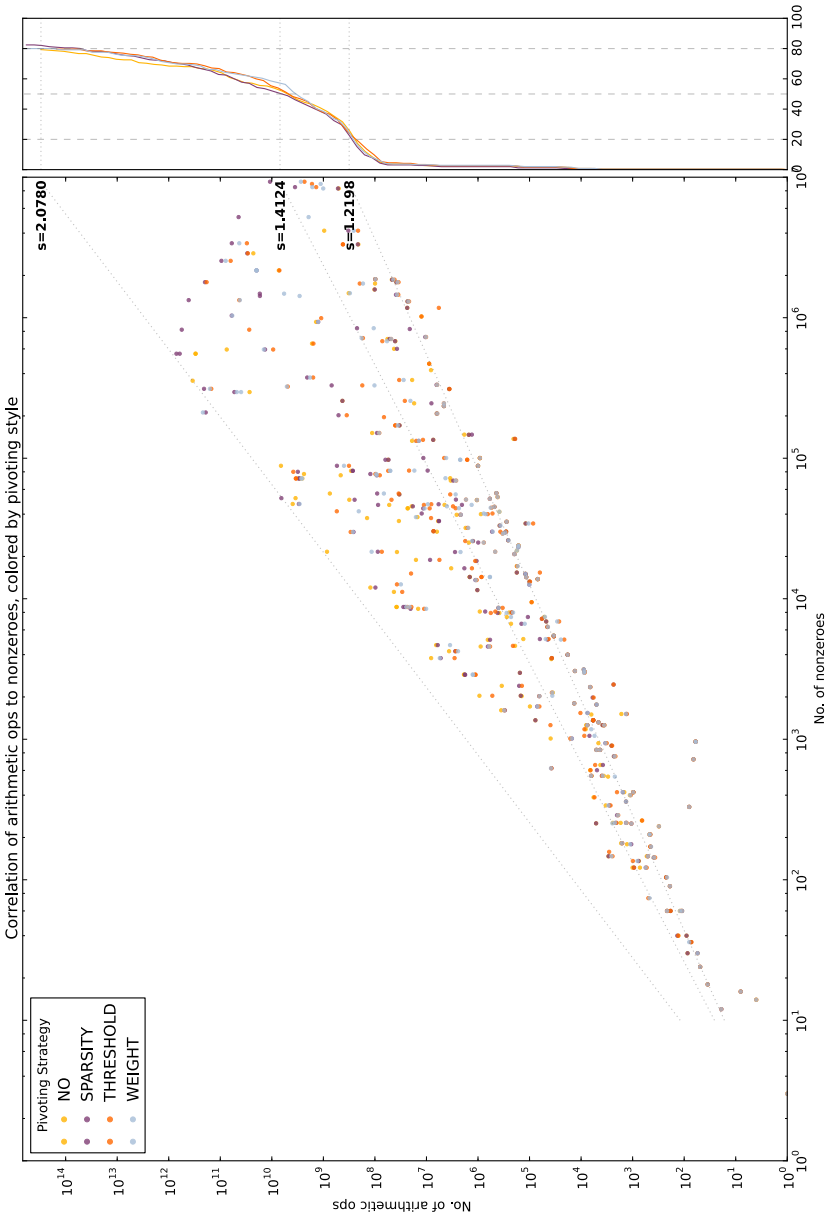


FIGURE 4.2. Scatter plot of the total number of arithmetic operations performed in elimination (y-axis) vs. number of nonzero entries in a matrix (x-axis). The small plot on the right shows the percentage (x-axis) of points in the plot lying under the line of a given slope (y-axis). The three oblique dotted lines mark the slopes such that 20%, 50%, and 80% of the SIMC matrices lie below the line. (Matrices on which "Rheinfall" failed are considered to take infinite time.)

Matrix	Rows	Columns	Nonzero	Rheinfall	LinBox
M0,3-D2	3	4	6	< 4ms	< 4ms
M0,4-D3	20	99	156	< 4ms	< 4ms
M0,4-D4	99	144	420	< 4ms	< 4ms
M0,4-D5	144	64	264	< 4ms	< 4ms
M0,5-D4	210	2112	3720	< 4ms	< 4ms
M0,5-D5	2112	7260	23280	0.004	0.050
M0,5-D6	7260	11280	50760	0.032	0.640
M0,5-D7	11280	8160	46560	0.040	1.030
M0,5-D8	8160	2240	15360	0.007	0.130
M0,6-D5	3024	49800	95760	0.005	0.160
M0,6-D6	49800	294480	1020240	0.321	57.200
M0,6-D7	294480	862290	4162320	5.285	4362.610
M0,6-D8	862290	1395840	8498160	53.787	40709.120
M0,6-D9	1395840	1274688	9286560	71.050	94797.670
M0,6-D10	1274688	616320	5201280	99.694	38086.330
M0,6-D11	616320	122880	1175040	2.312	1144.300
M1,2-D3	3	10	10	< 4ms	< 4ms
M1,2-D4	10	15	14	< 4ms	< 4ms
M1,2-D5	15	9	16	< 4ms	< 4ms
M1,3-D4	54	408	732	< 4ms	< 4ms
M1,3-D5	408	1112	3720	< 4ms	< 4ms
M1,3-D6	1112	1440	6636	0.003	0.010
M1,3-D7	1440	918	5166	0.002	0.010
M1,3-D8	918	236	1500	< 4ms	< 4ms
M1,4-D5	1008	13000	28464	0.002	0.030
M1,4-D6	13000	63756	245808	0.078	3.530
M1,4-D7	63756	160128	841872	1.599	176.500
M1,4-D8	160128	227564	1482996	40.169	2125.450
M1,4-D9	227564	185760	1427700	61.862	2758.020
M1,4-D10	185760	81504	716352	6.818	656.350
M1,4-D11	81504	14944	146832	0.097	11.610
M2,1-D4	3	20	22	< 4ms	< 4ms
M2,1-D5	20	39	122	< 4ms	< 4ms
M2,1-D6	39	43	179	< 4ms	< 4ms
M2,1-D7	43	28	136	< 4ms	< 4ms
M2,1-D8	28	9	40	< 4ms	< 4ms
M2,2-D5	97	1057	2451	< 4ms	< 4ms
M2,2-D6	1057	4519	18573	0.012	0.040
M2,2-D7	4519	10077	56090	0.407	0.500
M2,2-D8	10077	12927	88343	2.880	2.230
M2,2-D9	12927	9681	77218	1.116	2.230
M2,2-D10	9681	3983	35738	0.059	0.540
M2,2-D11	3983	713	6888	0.003	0.030

TABLE 4.2. CPU times (in seconds) for computing the matrix rank of the $\mathcal{M}_{g,n}$ homology matrices. The "Rheinfall" column reports times for the sample C++ implementation. The "LinBox" column reports times for the GEPP implementation in LINBOX version 1.1.7. Results were obtained on the UZH cluster "idhydra", featuring 48 Intel Xeon CPUs X7542 @ 2.67GHz, running 64-bit SuSE Linux Enterprise Server 11; both programs were compiled with GCC 4.3.4 with options `-O2 -march=native`.

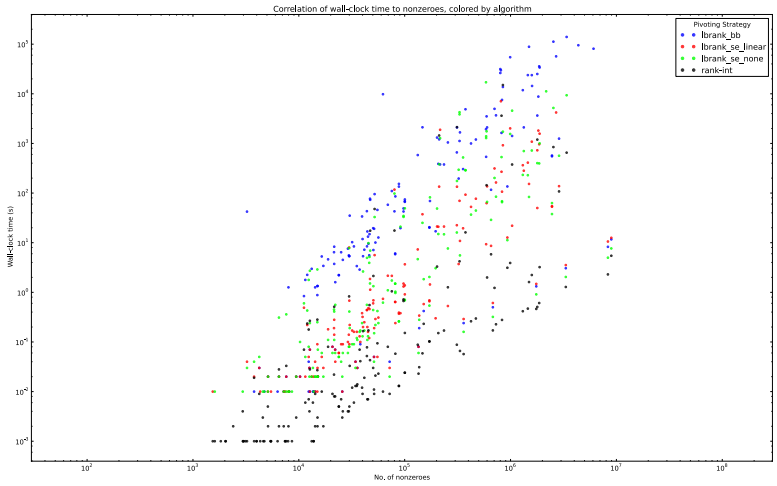


FIGURE 4.3. Scatter plot of the running time of the sequential Rheinfall variant and the three different rank-computation algorithms implemented in LinBox. The y-axis reports run time in seconds (log-scale); the x-axis (log-scale) shows the number of nonzero entries in each matrix.

MATRIX	Rows	Columns	Nonzero	Fill%	Rheinfall	LinBox
olivermatrix.2	78661	737004	1494559	0.0026	**2.68**	115.76
Trec14	3159	15905	2872265	5.7166	**116.86**	136.56
GL7d24	21074	105054	593892	0.0268	95.42	**61.14**
IG5-18	47894	41550	1790490	0.0900	1322.63	**45.95**

TABLE 4.3. CPU times (in seconds) for computing the matrix rank of selected integer matrices. The "Rheinfall" column reports times for the sample C++ implementation. The "LinBox" column reports times for the GEPP implementation in LinBox version 1.1.7. This is an extract from the table in Section 3, which see for the test setting and the characteristics of the hardware were it was run.

MATRIX	Rheinfall *(T)*	Rheinfall	LinBox *(T)*	LinBox
M0,6-D8	*No mem.*	**23.81**	50479.54	36180.55
M0,6-D10	**37.61**	23378.86	26191.36	13879.62
olivermatrix.2	**0.72**	2.68	833.49	115.76
Trec14	*No mem.*	116.86	**43.85**	136.56
GL7d24	**4.81**	95.42	108.63	61.14
IG5-18	12303.41	1322.63	787.05	**45.95**

TABLE 4.4. CPU times (in seconds) for computing the matrix rank of selected integer matrices and their transposes; the table compares running times of the Rheinfall/C++ and GEPP LinBox 1.1.7 codes. The columns marked with *(T)* report CPU times used for the transposed matrix; boldface font marks the fastest execution on each table row. Computation of the transposes of matrices "M0,6-D8" and "Trec14" exceeded the available 24 GB of RAM. Hardware, compilation flags and running conditions as in Section 3.

MATRIX	N	nonzero	fill%	Rheinfall	SuperLU
bbmat	38744	1771722	0.118	83.37	**1756.84**
g7jac200sc	59310	837936	0.023	87.69	**1722.28**
lhr71c	70304	1528092	0.030	*No mem.*	**926.34**
mark3jac140sc	64089	399735	0.009	92.67	**1459.39**
torso1	116158	8516500	0.063	97.01	**1894.19**
twotone	120750	1224224	0.008	91.62	**1155.53**

TABLE 4.5. Average Mflop/s attained in running LU factorization of square $N \times N$ matrices. The table compares the performance of the sample Rheinfall/C++ LU factorization with SUPERLU 4.2. The test matrices are a subset of those used in [26]. Hardware, compilation flags and running conditions as in Section 3.

Threads	Run time (s)					
	GL7d24	IG5-18	M0,6-D8	M0,6-D9	M0,6-D10	M0,6-D11
1	83.407	755.613	23.580	43.585	38.964	1.360
2	50.399	423.672	19.767	36.105	34.182	0.923
3	35.431	301.947	15.146	28.434	25.168	0.831
4	25.573	242.840	12.129	23.588	22.568	0.629
6	17.081	167.310	8.323	17.713	19.249	0.329
8	12.147	135.343	9.840	20.978	24.715	1.064
12	9.402	185.682	25.686	51.051	63.390	2.185
16	8.950	136.516	20.273	46.687	54.476	1.997
24	9.006	150.430	23.403	87.307	63.764	0.387

TABLE 4.7. Performance data relative to the sample implementation of "Rheinfall" in the shared-memory "coarse grained" variant on the set of SIMC test matrices.

Threads	Run time (s)			
	I10000	I100000	I1000000	I10000000
1	0.004	0.053	0.635	6.608
2	0.003	0.027	0.275	2.837
3	0.002	0.022	0.222	2.295
4	0.002	0.019	0.195	1.944
6	0.002	0.017	0.169	1.686
8	0.002	0.089	0.828	8.437
12	0.011	0.131	1.974	19.243
16	0.015	0.149	1.208	14.660
24	0.021	0.153	1.507	13.352

TABLE 4.9. Performance data relative to the sample implementation of "Rheinfall" in the shared-memory "coarse grained" variant on a set of identity matrices of varying size.

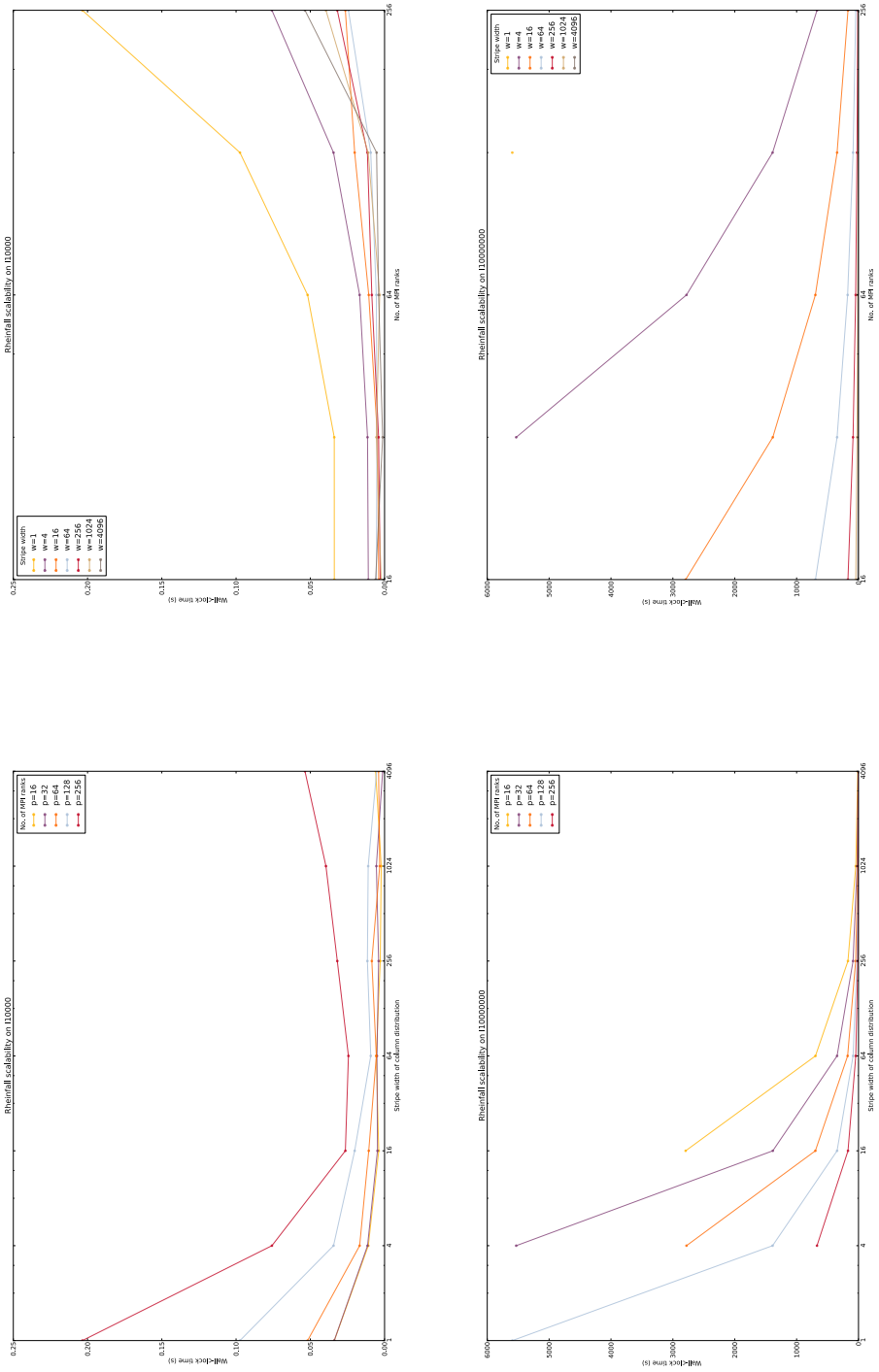


FIGURE 4.4. Running time (in seconds, y-axis) of the sample implementation of "Rheinfall" in the distributed-memory variant on identity square matrices of size 10^4 (top row) and 10^7 (bottom row), plotted by number of MPI ranks (left column) or stripe width (right column). Since no elimination work is performed, this measures the performance of the underlying MPI messaging system.

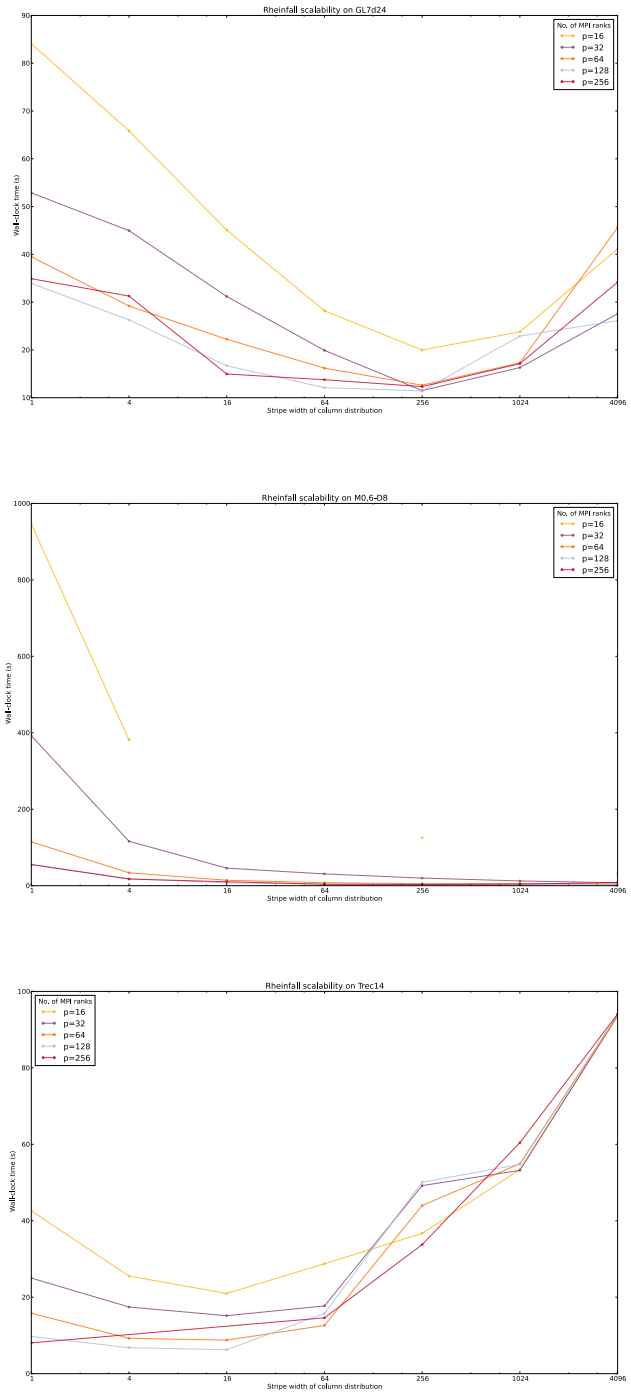


FIGURE 4.5. Plot of the running time (in seconds, y-axis) of the sample "Rheinfall" distributed-memory implementation on three different matrices, versus the number p of MPI ranks (x-axis); colored lines show different values of the stripe width w. The three example matrices are (top to bottom): GL7d24 (SIMC group GL7), M0,6-D8 (group Mgn), Trec14 (group Kocay).

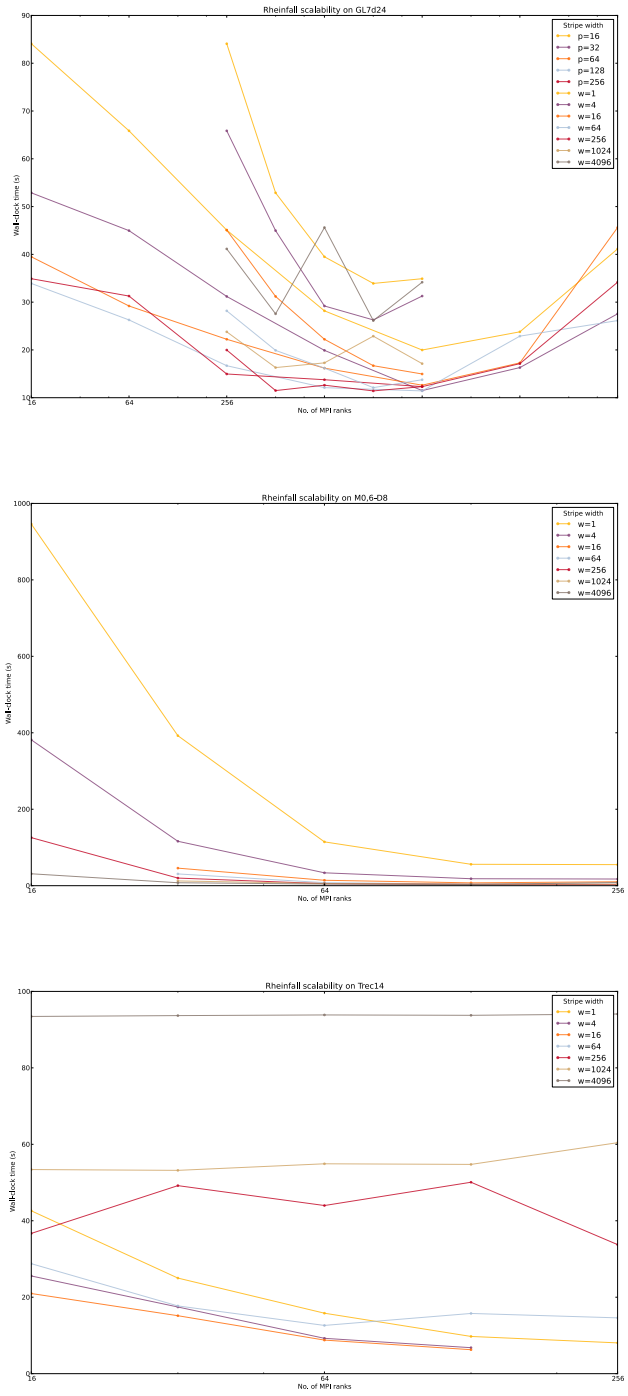


FIGURE 4.6. Plot of the running time (in seconds, y-axis) of the sample "Rheinfall" distributed-memory implementation on three different matrices, versus the stripe width w (x-axis); colored lines show different numbers of MPI ranks p. The three example matrices are (top to bottom): GL7d24 (SIMC group GL7), M0,6-D8 (group Mgn), Trec14 (group Kocay).

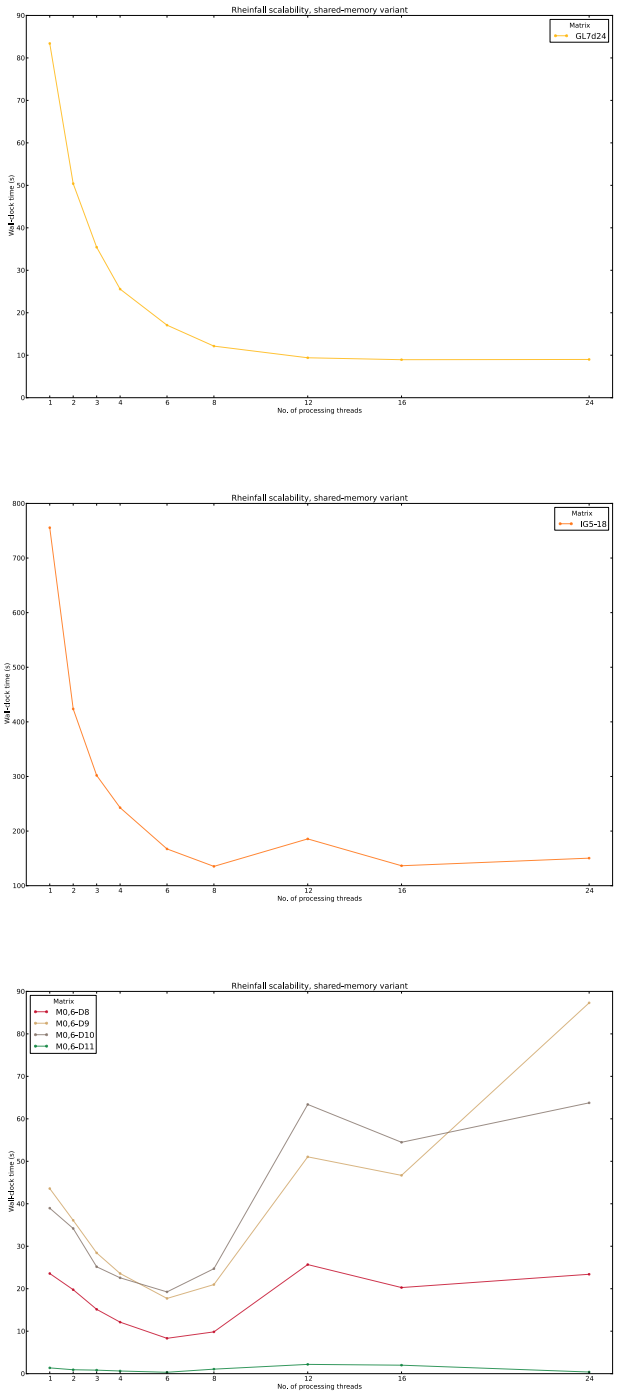


FIGURE 4.7. Scalability and performance of the sample implementation of "Rheinfall" in the shared-memory "coarse grained" variant.

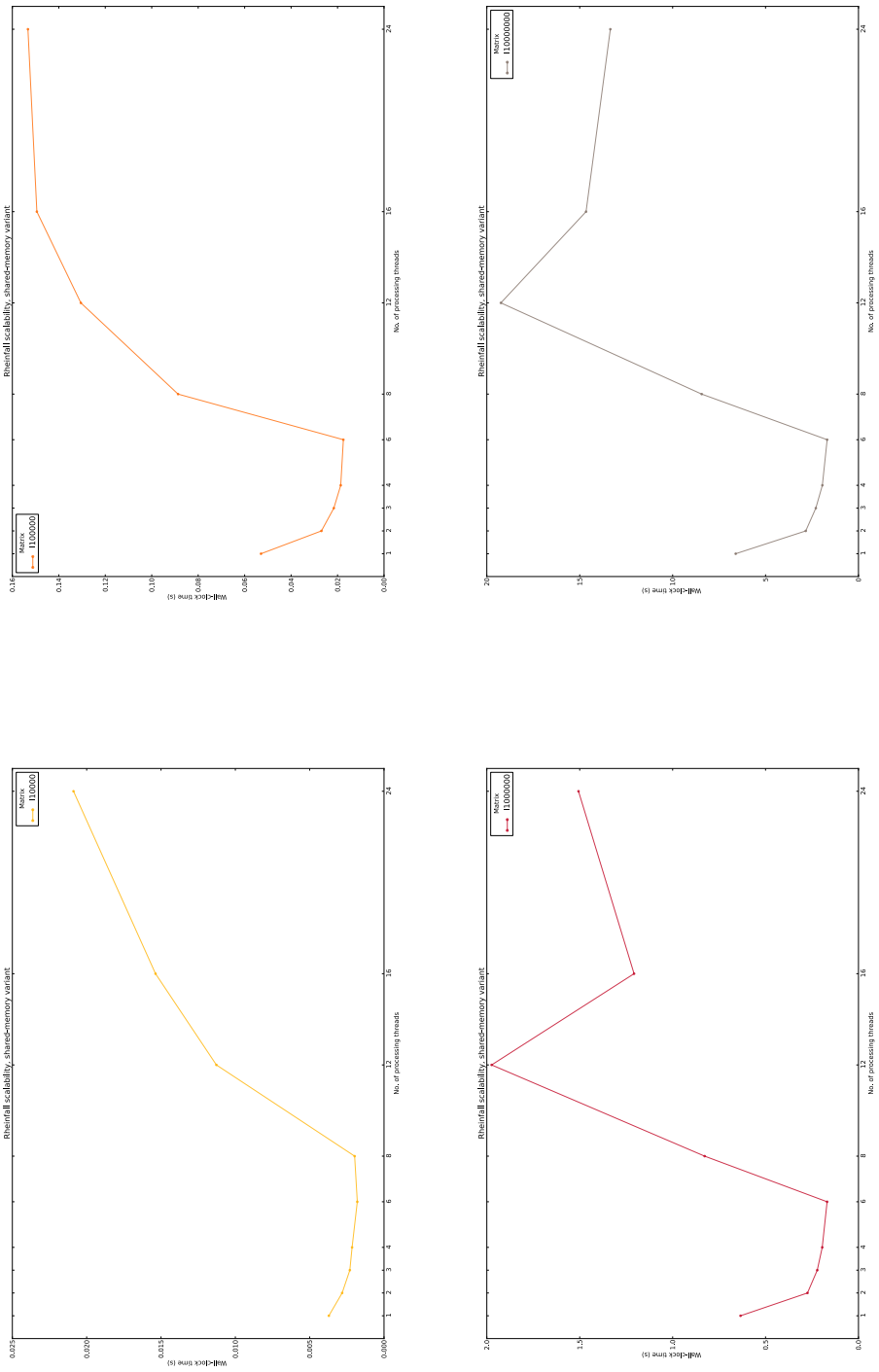


FIGURE 4.8. Performance of the sample implementation of "Rheinfall" in the shared-memory "coarse grained" variant on identity matrices of different sizes. Since no elimination work is performed, this measures the latency and overhead of the underlying TBB library.

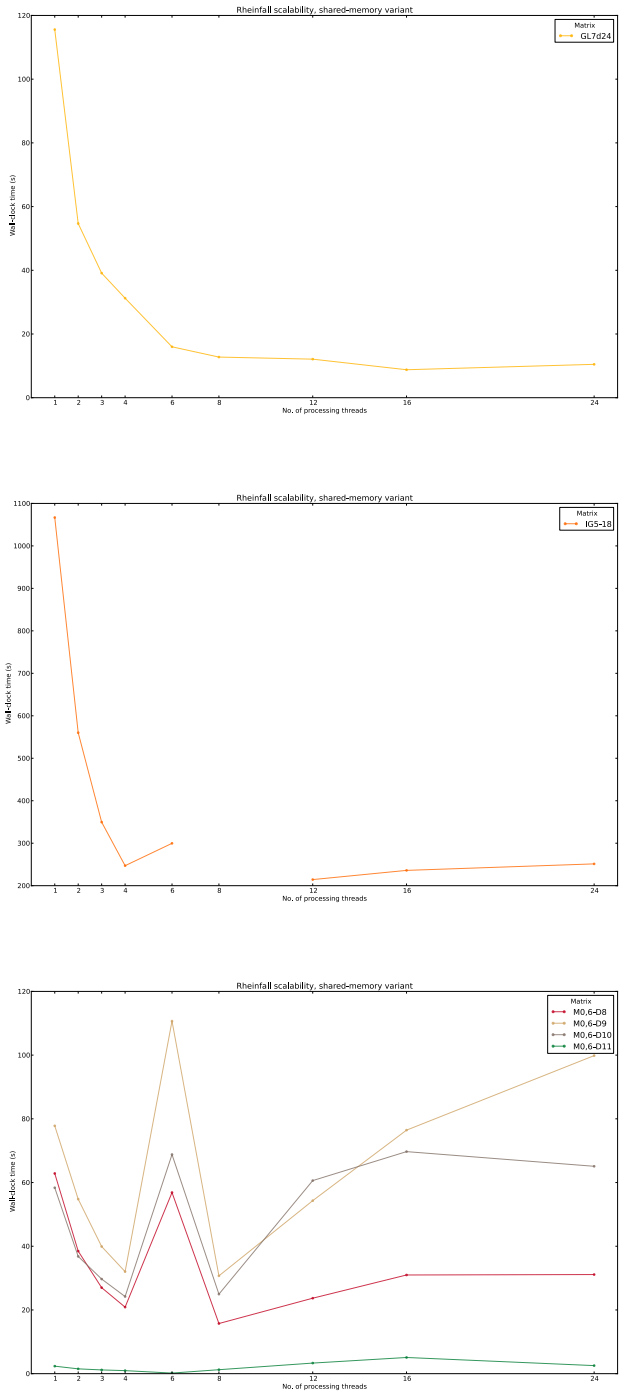


FIGURE 4.9. Scalability and performance of the sample implementation of "Rheinfall" in the shared-memory "fine grained" variant.

Threads	Run time (s)					
	GL7d24	IG5-18	M0,6-D8	M0,6-D9	M0,6-D10	M0,6-D11
1	115.572	1066.811	62.853	77.806	58.368	2.356
2	54.679	560.156	38.464	54.786	36.809	1.510
3	39.108	349.644	26.998	39.920	29.704	1.176
4	31.246	247.078	20.882	32.030	24.215	0.954
6	15.981	299.733	56.861	110.688	68.791	0.184
8	12.758	*No data*	15.740	30.705	24.962	1.244
12	12.103	214.365	23.690	54.310	60.609	3.324
16	8.780	236.029	30.979	76.448	69.706	5.081
24	10.476	251.333	31.115	99.841	65.100	2.532

TABLE 4.11. Performance data relative to the sample implementation of "Rheinfall" in the shared-memory "fine grained" variant on the set of SIMC test matrices.

Threads	Run time (s)			
	I10000	I100000	I1000000	I10000000
1	0.005	0.064	0.734	7.614
2	0.002	0.016	0.161	1.720
3	0.001	0.013	0.134	1.398
4	0.001	0.012	0.250	1.262
6	0.001	0.010	0.103	1.058
8	0.002	0.014	0.107	1.085
12	0.001	0.020	0.103	0.965
16	0.016	0.092	1.561	4.912
24	0.004	0.032	2.249	18.715

TABLE 4.13. Performance data relative to the sample implementation of "Rheinfall" in the shared-memory "fine grained" variant on a set of identity matrices of varying size.

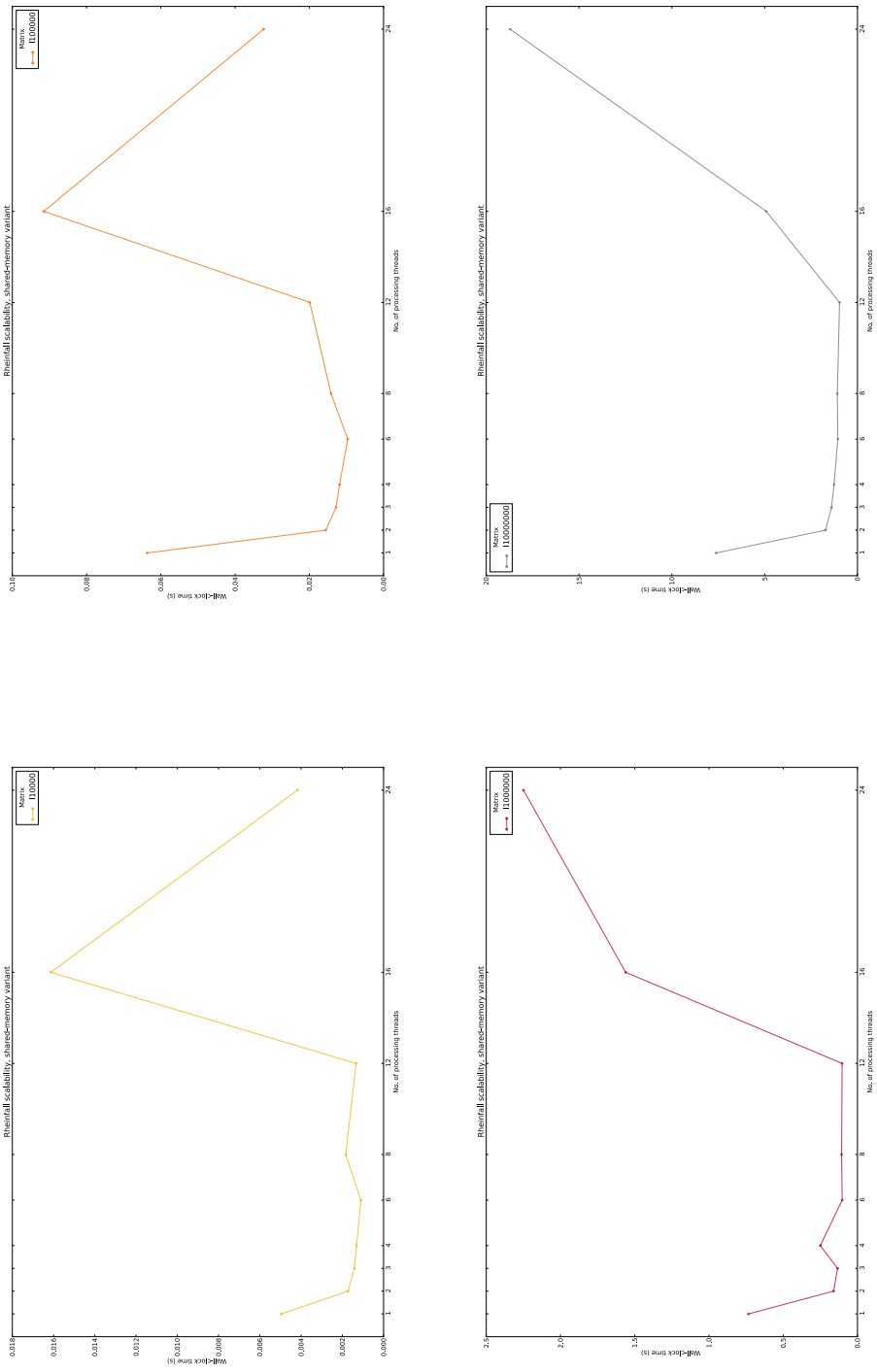


FIGURE 4.10. Performance of the sample implementation of "Rheinfall" in the shared-memory "fine grained" variant on identity matrices of different sizes. Since no elimination work is performed, this measures the latency and overhead of the underlying TBB library.

p	w	MPI Total% Avg.$\pm\sigma$	MPI Total% Max.	MPI Total% Min.	MPI_Recv% Avg.	MPI_Recv% Max.	MPI_Iprobe% Avg.	MPI_Iprobe% Max.	MPI_Barrier% Avg.	MPI_Barrier% Max.
16	16	18.79 ± 0.32	19.61	18.37	79.96	81.51	3.13	3.29	0.00	0.00
32	16	11.54 ± 0.53	12.87	10.93	4.41	71.97	14.19	14.97	0.00	0.00
64	16	12.25 ± 0.20	12.78	11.77	1.12	55.94	34.57	36.13	0.00	0.00
128	16	20.51 ± 0.55	23.87	19.33	31.62	35.06	61.08	64.17	0.00	0.00
16	64	13.78 ± 0.35	14.43	13.28	79.24	81.08	3.86	4.26	0.00	0.00
32	64	10.11 ± 0.31	10.60	9.33	73.49	76.38	14.45	15.87	0.00	0.00
64	64	8.10 ± 0.19	8.62	7.72	34.13	37.35	56.15	60.29	0.00	0.21
128	64	16.76 ± 0.40	17.80	15.79	17.32	23.52	78.16	83.47	0.00	0.16
256	64	38.32 ± 0.90	40.70	36.05	4.00	6.65	94.00	96.53	0.00	0.11
16	256	26.77 ± 1.79	29.50	23.29	89.69	92.31	1.27	1.50	0.00	0.20
32	256	10.17 ± 1.34	13.57	7.83	78.51	85.33	13.10	18.08	0.00	0.01
64	256	16.55 ± 2.16	22.15	11.74	61.46	73.13	27.20	43.73	0.00	0.67
128	256	15.43 ± 0.64	18.65	14.35	6.15	10.97	90.98	95.27	0.00	0.02
256	256	38.92 ± 1.94	43.24	32.99	6.12	14.20	89.38	97.41	0.00	0.60
16	1024	13.90 ± 1.21	16.78	11.40	79.13	87.79	3.94	4.51	0.00	0.01
32	1024	9.28 ± 1.82	14.29	5.91	1.07	82.24	16.53	24.06	0.00	0.05
64	1024	7.43 ± 0.76	10.69	6.26	28.30	54.48	62.95	76.12	0.00	0.25
128	1024	14.60 ± 1.52	18.17	11.31	9.84	25.45	82.82	94.65	2.05	4.90
256	1024	27.62 ± 3.87	38.58	18.22	1.92	8.65	91.02	99.08	4.95	15.25
16	4096	9.08 ± 1.62	13.81	7.22	50.57	66.97	7.52	10.35	0.00	0.19
32	4096	6.53 ± 1.97	12.33	3.71	36.80	58.48	28.30	51.23	1.51	3.71
64	4096	6.81 ± 1.52	12.01	4.53	8.73	30.15	72.13	93.92	10.88	26.93
128	4096	16.73 ± 7.80	44.72	8.59	5.12	21.82	46.82	92.24	43.69	86.65
256	4096	45.78 ± 28.32	88.22	9.91	0.00	9.18	11.94	96.68	86.09	98.63

TABLE 4.14. Percentage of running time spent in MPI communication for the sample Rheinfall/C implementation on the matrices M0,6-Dk, with varying number of MPI ranks and stripe width parameter w. Columns `MPI_Recv`, `MPI_Iprobe` and `MPI_Barrier` report on the percentage of MPI time spent spent servicing these calls; in these cases, the minimum is always very close to zero, hence it is omitted from the table. Tests were executed on the UZH cluster "Schroedinger"; see Table 4.3 for hardware details. The MPI layer was provided by OpenMPI version 1.4.3, using the TCP/IP transport.

APPENDIX A

Pseudo-code notation

Blocks of code are marked by indentation (rather than delimited by specific keywords).

The 'def' keyword is used to mark the beginning of a function definition.

The notation 'for x in S' is used to loop over all the items x in a set or sequence S; sometimes the notation 'for x in $a, \ldots, b$' is used instead. The form 'for i, x in *enumerate*(S)' is used for keeping track of the iteration number when looping over the elements of S: as x runs over the items in S, i orderly takes the values 0, 1, ..., up to $|S| - 1$.

1. Basic types

Numbers and basic data structures (arrays, lists, sets; see below) are considered basic types, together with the logical constants *True* and *False*, and the special value *None*.

2. Objects

The word "object" is used to denote a kind of aggregate type: an *object* is a tuple '(a_1, a_2, ..., a_N)', where each of the slots a_i can be independently assigned a value; the values assigned to different a_i's need not be of the same type. We write $X.a_i \leftarrow b$ to mean that the slot a_i of object X is assigned the value b. Unless otherwise noted, object slots are mutable, i.e., they can be assigned different values over the course of time.

An objects' *class* is the tuple '(a_1, ..., a_N)' of slots names that defines the object; the actual tuple of values is called an *instance*.[1]

3. Arrays, lists, sets

A few types of basic data structures are used in the code: arrays, lists and sets. They are distinguished only for clarity, and we make no assumption that these are primitive: for instance, each of these data structures could be implemented on top of the "list" type defined here.

An "array" is a fixed-size collection of elements of the same type; the number and type of elements stored in an array will be stated when the array is first created.

[1] Readers familiar with object-oriented programming will note that this is an over-simplified version of the usual object-oriented definition of objects and classes; this originates in the fact that the concrete implementation of the algorithms was done in object-oriented Python, but object-orientation is by no means essential to the implementation.

Items in an array can be accessed by position: if a is an array, then its k-th element will be accessed as $a[k]$. Array elements can be mutated; we write $a[k] \leftarrow b$ to mean that object b is stored into the k-th place of array a.

A "list" is a variable-size collection of objects. Two features distinguish lists from arrays: *(1)* lists can grow and shrink in size, and *(2)* lists can store items of different types. If l is a list with n elements, the notation $l.append(x)$ will be used to mean that x should be added as $(n+1)$-th item in list l. Again, the square bracket notation $l[k]$ is used to denote the value stored in the k-th place in l, and $l[k] \leftarrow x$ means that the k-th slot of l is mutated to the value x. The operator "+" stands for concatenation when applied to lists.

A "set" is a mutable unordered collection of objects of the same type. The only relevant difference with sets in the mathematical sense of the word is that set variables are mutable: if s is a set, then $s.add(x)$ will be used to specify that s should be mutated into the set $s \cup \{x\}$. No duplicates are admitted: if $x \in s$ and $x = y$, then $s.add(y)$ does not alter s in any way.

The word "sequence" will be used to denote any one of the above three. When S is a sequence, we define $size(S)$ as the number of elements in S; if S is a *list* or *array* object, valid indices into S range from 0 to $size(S)-1$.

3.1. List comprehensions. A special syntax is used to form a list when its items can be gotten by applying a function or operation to the elements of another sequence.

The notation '$L \leftarrow [f(x)$ for x in $S]$' makes L into the list formed by evaluating function f on each element in S, analogously to the usual notation $\{f(x) : x \in S\}$ for sets.

As an extension, the expression '$L \leftarrow [f(x)$ for x in S if $P(x)]$' makes L into the list of values of f over the set S' of elements of S for which the predicate $P(x)$ is true: $S' = \{x : x \in S \wedge P(x)\}$.

4. Operators

The "%" operator is used to take the remainder of integer division: for integers k and $n > 0$, the expression $(k \ \% \ n)$ evaluates to the smallest non-negative residue of $k \mod n$.

The "+" operator normally denotes addition when applied to numbers, and concatenation when applied to lists.

Any other operator keeps it usual mathematical meaning.

APPENDIX B

Comparison of fatgraph generation methods

This section compares three different approaches to generating trivalent fatgraphs: namely, we compare the *MgnTrivalentGraphs* algorithm described in Section 3.1.3 with two alternatives.[1] Table B.1 presents a summary of results.

None of the suggested algorithms is capable of directly producing an isomorph-free set of distinct fatgraphs; they all produce a larger set of fatgraphs that must be reduced by taking only one representative per isomorphism class of fatgraphs. Therefore, Table B.1 also reports the actual number of distinct fatgraphs for a given g, n pair; not all counts are known: a cell is left empty when the corresponding count has not yet been computed. From the results gathered so far, it is apparent that *all* algorithms overestimate the actual number of fatgraphs.

g	n	N	N_1^+	N_2^+	N_3
0	3	2	–	15	5 760
0	4	6	84	630	1.072964×10^{13}
0	5	26	936	15 015	4.593811×10^{24}
0	6	191	8 892	306 306	6.326929×10^{37}
0	7		114 600	5 819 814	1.132261×10^{52}
1	1	1	–	15	5 760
1	2	5	114	630	1.072964×10^{13}
1	3	46	1 644	15 015	4.593811×10^{24}
1	4	669	24 156	306 306	6.326929×10^{37}
1	5		511 416	5 819 814	1.132261×10^{52}
2	1	9	6 336	15 015	4.593811×10^{24}
2	2	368	17 982	306 306	6.326929×10^{37}
2	3		606 144	5 819 814	1.132261×10^{52}
3	1		1 065 718 368†	5 819 814	1.132261×10^{52}

TABLE B.1. Number of (non-unique) trivalent fatgraphs generated according to different algorithms. The N column reports the actual number of distinct fatgraphs for the given g, n; empty cells mean the corresponding number has not been computed. The N_1^+ column lists upper bounds for the recursive generation algorithm *MgnTrivalentGraphs* (see Listing 7); values marked with the "†" symbol are estimated using earlier values of N_1^+ because the corresponding values of N are not available. The N_2^+ values bound from above the number of fatgraphs generated by grafting binary trees into clovers. Finally, N_3 is the count of fatgraphs generated by enumerating pairs of permutations (as per combinatorial definition of fatgraph).

[1] The author is aware of no other algorithm for generating the set of all fatgraphs. The comparison here is taken with the solutions used in earlier attempts of implementation of the FatGHoL software.

In what follows, let $N(g,n) := |\mathcal{R}_{g,n}|$ be the number of distinct (g,n)-fatgraphs; also define:

$$\begin{aligned}\xi(g,n) &:= 2g+n,\\ m_{\max}(g,n) &:= 6g+3n-6 = 3\xi-6,\\ m_{\min}(g,n) &:= 2g+n-1 = \xi-1.\end{aligned}$$

It is trivial to check that $m_{\max}$ and $m_{\min}$ are the maximum and minimum number of edges that a (g,n)-fatgraph can have.

1. Generation by recursive edge addition

The algorithm *MgnTrivalentGraphs* described in Section 3.1.3 produces a (g,n)-fatgraph by adding an edge to fatgraphs with lower (g,n); the procedure can then be applied recursively.

Let $N_1(g,n)$ be the number of (non distinct) fatgraphs returned by function *MgnTrivalentGraphs*(g,n). According to Section 3.1.3, this can be written as:

$$N_1(g,n) = N_{1,A}(g,n) + N_{1,B}(g,n) + N_{1,C}(g,n),$$

where $N_{1,A}$, $N_{1,B}$, $N_{1,C}$ are the numbers of fatgraphs constructed in cases *A)*, *B)*, *C')* of Algorithm 7.

In case *A)*, we have 1 generated (g,n)-fatgraph per each pair formed by a $(g,n-1)$-fatgraph and one of its *oriented* edges, modulo the action of the automorphism group $\mathrm{Aut}(G)$. However, we do not know how to estimate the number of orbits of this $\mathrm{Aut}(G)$-action. Since the generic fatgraph only has one automorphism, an upper bound can instead be given by considering all pairs formed by a fatgraph and an oriented edge:

$$N_{1,A}(g,n) \leqslant N^+_{1,A}(g,n) := 2\cdot m_{\max}(g,n-1)\cdot N(g,n-1).$$

In case *B)*, the algorithm generates one (g,n)-fatgraph per each triplet formed by a $(g,n-1)$-fatgraph and two oriented edges, not necessarily distinct (modulo the action of $\mathrm{Aut}\, G$); a similar remark about the upper bound applies:

$$N_{1,B}(g,n) \leqslant N^+_{1,B}(g,n) := (2\cdot m_{\max}(g,n-1))^2\cdot N(g,n-1)$$

In case *C')*, the computation is exactly the same, except we apply the q construction to fatgraphs belonging in $\mathcal{R}_{g-1,n+1}$:

$$N_{1,C}(g,n) \leqslant N^+_{1,C}(g,n) := 4\cdot m_{\max}(g-1,n+1)^2\cdot N(g-1,n+1).$$

Table B.1 shows the upper bound given by:

$$N^+_1(g,n) := N^+_{1,A}(g,n) + N^+_{1,B}(g,n) + N^+_{1,C}(g,n).$$

According to Table B.1, the *MgnTrivalentGraphs* algorithm outperforms the alternative procedures when $2g+n<7$, and apparently generates a much larger set of fatgraphs when $2g+n>7$. However, the values were obtained using $N^+_1(g,n)$ instead of $N(g,n)$ in recursive computations when the actual value of $N(g,n)$ is not known; therefore $N^+_1(g,n)$ might grossly overestimate the number of graphs considered by *MgnTrivalentGraphs* for $2g+n>6$. Further investigation is needed to ascertain whether this is due to the algorithm of Section 2 being asymptotically faster, or to the estimate for $N_1(g,n)$ being grossly imprecise when no data about

the real number of trivalent fatgraphs in the recursion step is known. However, the author conjectures that this estimate holds:

$$N_1(g,n) \leqslant O(\xi^3) \cdot N(g,n) \tag{1.1}$$

2. Generation by insertion of binary trees

A different approach is the following:

- Generate all distinct (g,n)-fatgraphs with 1 vertex; each such fatgraph has $m_{\min}(g,n)$ edges, hence the vertex has valence $2 \cdot m_{\min}(g,n)$.
- Given any such fatgraph G_0, build a trivalent (g,n)-fatgraph G by replacing the vertex with a full binary tree on $2 \cdot m_{\min}(g,n)$ leaves.

Call a fatgraph with only one vertex a *clover*. Let $N_2'(g,n)$ be the number of distinct (g,n)-clovers; we can estimate it as follows.

Lemma B.1. *The number of isomorphic clovers is equal to the number of orbits of the adjoint action of* $(12\dots 2m)$ *over the set of self-conjugate permutations* $\{\sigma_1 \in \mathfrak{S}_{2m} : \sigma_1^2 = \mathrm{id}\}$.

Proof. Let $G_0 = (L; \sigma_0, \sigma_1, \sigma_2)$ be a (g,n)-fatgraph given in combinatorial form, where $L = \{1,\dots,2m\}$ and $\sigma_i \in \mathfrak{S}_m$. If G_0 is a clover, then σ_0 is a permutation formed by just one cycle; without loss of generality we may assume σ_0 is the rotation $(12\dots 2m)$. Let $G_0' = (L; \sigma_0', \sigma_1', \sigma_2)$ be another (g,n)-clover: by the same reasoning we have $\sigma_0' = \sigma_0 = (12\dots m)$; if $f\colon G_0 \to G_0'$ is an isomorphism, then f commutes with σ_0 hence $f = \sigma_0^j$ for some $j|2m$. Therefore, from $\sigma_1 \circ f = f \circ \sigma_1'$ we get $\sigma_1' = \sigma_0^{-j} \circ \sigma_1 \circ \sigma_0^j$. This proves the claim. □

Lemma B.2. *Let L be a finite set of $l = p{\cdot}q$ elements. The number of permutations of L which can be expressed as product of q disjoint p-cycles is:*

$$C(p,q) = \prod_{i=1}^{q} \prod_{j=1}^{p-1} (pi - j). \tag{2.1}$$

Proof. Without loss of generality we can assume $L = \{1,\dots,pq\}$; let $\tau \in \mathfrak{S}_{pq}$ be a permutation composed of q disjoint p-cycles. We can give a "canonical" form to τ if we order its cycles by stipulating that:

- a cycle $(a_1 a_2 \dots a_p)$ is always written such that $a_1 = \min a_i$;
- $(a_1 a_2 \dots a_p)$ precedes $(b_1 b_2 \dots b_p)$ iff $\min a_i < \min b_i$.

Now assume τ is written in this canonical form; then $a_1 = 1$ and we have $pq-1$ choices for the element $a_2 = \tau(a_1)$ following a_1 in the cycle, $pq-2$ choice for the next element $a_3 = \tau(a_2)$, and so on until the final element a_p of the first cycle. Then starting element a_{p+1} of the second cycle has to be the minimum element of $L \setminus \{a_1, a_2, \dots, a_p\}$, but we have $(p-1)q-1$ choices for $a_{p+2} = \tau(a_{p+1})$: an iterative argument proves the assertion. □

Lemma B.3. *The number of distinct self-conjugate permutations on a set of l elements is given by* $(l-1)!! := (l-1) \cdot (l-3) \cdot \ldots \cdot 1$.

Proof. A self-conjugate permutation τ on a set L of $l = 2m$ elements is the product of m disjoint 2-cycles, and the the result follows from Lemma B.2. □

Combining Lemma B.1 and B.3, we immediately get the following estimate:

$$\frac{(2m-1)!!}{2m} \leqslant N_2'(g,n) \leqslant (2m-1)!!, \qquad m = m_{\min}(g,n),$$

where the upper bound comes from assuming that no two clovers can be transformed one into the other by a rotation, and the lower bound comes from considering all clovers as part of the same equivalence class.

In order to create a trivalent fatgraph from a clover, we replace the vertex with a full binary tree with $l = 2m$ leaves; equivalently, we identify the leaves of the tree according to the same "gluing pattern" that identifies half-edges in the clover.

More precisely, let $G_0 = (L; \sigma_0, \sigma_1, \sigma_2)$ be a clover, with $L = \{1, \dots, 2m\}$ and $\sigma_0 = (12 \dots 2m)$ as above. Let L' be set of leaves of a chosen binary tree T and $f\colon L' \to L$ a bijection. Now $\tau := f^{-1} \circ \sigma_1 \circ f$ is a fixed-point free involution on L': by identifying leaves of T according to τ, we get a trivalent fatgraph G, which we say is obtained by plugging T into G_0 (by means of f).

Given a permutation ϕ' on L', the map $f' = f \circ \phi'$ is a bijection and we have:

$$\tau' = {f'}^{-1} \circ \sigma_1 \circ f' = {\phi'}^{-1} \circ (f^{-1} \circ \sigma_1 \circ f) \circ \phi' = {\phi'}^{-1} \circ \tau \circ \phi',$$

which is an involution on L' conjugate to τ. Conversely, if $\sigma_1' = \phi^{-1} \circ \sigma_1 \circ \phi$ is conjugate to σ_1, then $f' = \phi \circ f\colon L' \to L$ is again a bijection, hence:

$$f^{-1} \circ \sigma_1' \circ f = (f^{-1} \circ \phi^{-1}) \circ \sigma_1 \circ (\phi \circ f) = {f'}^{-1} \circ \sigma_1 \circ f',$$

which is the involution defining the attachment map of T to G_0 by means of f'. Since any two involutions are conjugate, we can fix the map f once and for all binary trees with the same number of leaves, and only let the involution σ_1 (i.e., the clover G_0) vary.

Therefore $N_2(g,n) = N_2'(g,n) \cdot Y(m_{\min}(g,n))$, where $Y(l)$ is the count of full binary trees with l leaves. The number $Y(l)$ is given by the $(l-1)$-th Catalan number:

$$Y(l) = \frac{(2l-2)!}{(l-1)! \cdot l!}.$$

Hence from (2) we get:

$$N_2^-(g,n) \leqslant N_2(g,n) \leqslant N_2^+(g,n),$$

where:

$$\begin{aligned} N_2^-(g,n) &:= \frac{1}{2m} \cdot \frac{(4m-2)!}{(2m-2)!!(2m)!}, \\ N_2^+(g,n) &:= \frac{(4m-2)!}{(2m-2)!!(2m)!}, \\ m &:= m_{\min}(g,n). \end{aligned} \tag{2.2}$$

3. Generation from permutations

As in the previous section, represent a fatgraph G as $(L; \sigma_0, \sigma_1, \sigma_2)$ where $L = \{1, \dots, 2m\}$. Here we count the number of trivalent fatgraphs that are generated by naively constructing a fatgraph from its combinatorial definition.

If G is trivalent, then σ_0 is a product of disjoint 3-cycles; by Lemma B.2, the number of such σ_0 is:

$$C(3,k) = (l-1)(l-2) \cdot (l-4)(l-5) \cdot \ldots \cdot 2 \cdot 1, \qquad l = 2m = 3k \tag{3.1}$$

For each chosen σ_0, each choice of a self-conjugate permutation σ_1 gives rise to a trivalent (g,n)-fatgraph; by Lemma B.3 there are exactly $(2m-1)!!$ such choices. Therefore, we have:

$$N_3(g,n) = (2m-1)!! \cdot C(3,2m/3) = (2m-1)!! \cdot (2m-1)(2m-2)\cdot \\ \cdot(2m-4)(2m-5)\cdot \ldots \cdot 2 \cdot 1, \tag{3.2}$$

where $m = m_{\max}(g,n)$.

APPENDIX C

Fatgraphs of $\mathcal{M}_{0,4}$

This appendix is a complete catalog of all the fatgraphs with $g = 0$ and $n = 4$. It is provided as an example of capabilities of the algorithms described in Chapter 3, and as a demo of the features implemented in the software FatGHoL. Other fatgraphs catalogs are available from the FatGHoL website at http://code.google.com/p/fatghol/downloads/list.

There are a total of 21 undecorated fatgraphs in the Kontsevich graph complex of $\mathcal{M}_{0,4}$, originating 327 marked ones.

In the following, we denote $G_{m,j}$ the j-th graph in the set of undecorated fatgraphs with m edges; the symbol $G^{(k)}_{m,j}$ denotes the k-th inequivalent marking of $G_{m,j}$.

Fatgraph vertices are marked with lowercase latin letters "a", "b", "c", etc.; edges are marked with an arabic numeral starting from "1"; boundary cycles are denoted by lowercase greek letters "α", "β", etc.

Automorphisms are specified by their action on the set of vertices, edges, and boundary cycles: for each automorphism A_k, a table line lists how it permutes vertices, edges and boundary cycles relative to the identity morphism A_0. The automorphism table is printed only if the automorphism group is non-trivial.

Automorphisms that reverse the orientation of the unmarked fatgraph are indicated with a "†" symbol in the automorphism table; those that reverse the orientation of the marked fatgraphs are distinguished with a "‡" sign.

If a fatgraph is orientable, a "Markings" section lists all the inequivalent ways of assigning distinct numbers $\{0, \ldots, n-1\}$ to the boundary cycles; this is of course a set of representatives for the orbits of $\mathfrak{S}_n$ under the action of $\mathrm{Aut}(G)$.

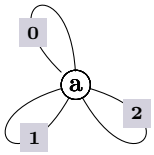
A separate section lists the differential of marked fatgraphs; graphs with null differential are omitted. If no marked fatgraph has a non-zero differential, the entire section is dropped.

Boundary cycles are specified using a "sequence of corners" notation: each corner is represented as ${}^{p}L^{q}$ where L is a latin letter indicating a vertex, and p, q are the attachment indices of the incoming and outgoing edges, respectively. Attachment indices match the Python representation of the vertex: e.g., if a=`Vertex([0,0,1])`, the two legs of edge 0 have attachment indices 0 and 1, and the boundary cycle enclosed by them is represented by the (single) corner ${}^{0}a^{1}$.

1. Fatgraphs with 3 edges / 1 vertex

There are 2 unmarked fatgraphs in this section, originating 40 marked fatgraphs (20 orientable, and 20 nonorientable).

1.1. The Fatgraph $G_{3,0}$ (8 orientable markings).




```
Fatgraph([
  Vertex([0, 0, 1, 1, 2, 2]),# a
])
```

1.1.1. *Boundary cycles.*

$$\alpha = ({}^{0}a^{1})$$
$$\beta = ({}^{1}a^{2} \to {}^{3}a^{4} \to {}^{5}a^{0})$$
$$\gamma = ({}^{2}a^{3})$$
$$\delta = ({}^{4}a^{5})$$

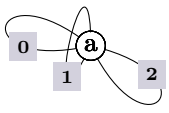
1.1.2. *Automorphisms.*

A_0	a	0	1	2	α	β	γ	δ
$A_1{}^{\ddagger}$	a	1	2	0	γ	β	δ	α
$A_2{}^{\ddagger}$	a	2	0	1	δ	β	α	γ

1.1.3. *Markings.*

	$G_{3,0}^{(0)}$	$G_{3,0}^{(1)}$	$G_{3,0}^{(2)}$	$G_{3,0}^{(3)}$	$G_{3,0}^{(4)}$	$G_{3,0}^{(5)}$	$G_{3,0}^{(6)}$	$G_{3,0}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

1.2. The Fatgraph $G_{3,1}$ (non-orientable, 12 orientable markings).




```
Fatgraph([
  Vertex([1, 0, 0, 1, 2, 2]),# a
])
```

1.2.1. *Boundary cycles.*

$$\alpha = ({}^{2}a^{3} \to {}^{0}a^{1})$$
$$\beta = ({}^{1}a^{2})$$
$$\gamma = ({}^{3}a^{4} \to {}^{5}a^{0})$$
$$\delta = ({}^{4}a^{5})$$

1.2.2. *Automorphisms.*

A_0	a	0	1	2	α	β	γ	δ
$A_1{}^{\dagger\ddagger}$	a	2	1	0	γ	δ	α	β

1.2.3. *Markings.*

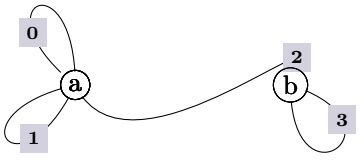
	$G_{3,1}^{(0)}$	$G_{3,1}^{(1)}$	$G_{3,1}^{(2)}$	$G_{3,1}^{(3)}$	$G_{3,1}^{(4)}$	$G_{3,1}^{(5)}$	$G_{3,1}^{(6)}$	$G_{3,1}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{3,1}^{(8)}$	$G_{3,1}^{(9)}$	$G_{3,1}^{(10)}$	$G_{3,1}^{(11)}$
α	1	1	2	2
β	2	3	0	1
γ	3	2	3	3
δ	0	0	1	0

2. Fatgraphs with 4 edges / 2 vertices

There are 6 unmarked fatgraphs in this section, originating 198 marked fatgraphs (99 orientable, and 99 nonorientable).

2.1. The Fatgraph $G_{4,0}$ (24 orientable markings).



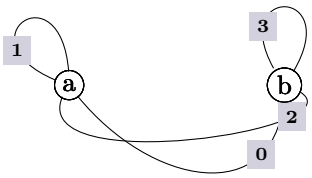

```
Fatgraph([
  Vertex([0, 0, 1, 1, 2]),# a
  Vertex([3, 3, 2]),      # b
])
```

2.1.1. *Boundary cycles.*

$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^2b^0 \to {}^4a^0 \to {}^3a^4 \to {}^1b^2)$$
$$\gamma = ({}^2a^3)$$
$$\delta = ({}^0b^1)$$

2.1.2. *Markings.* Fatgraph $G_{4,0}$ only has the identity automorphism, so the marked fatgraphs $G_{4,0}^{(0)}$ to $G_{4,0}^{(24)}$ are formed by decorating boundary cycles of $G_{4,0}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

2.2. The Fatgraph $G_{4,1}$ (12 orientable markings).




```
Fatgraph([
  Vertex([1, 1, 2, 0]),# a
  Vertex([0, 2, 3, 3]),# b
])
```

2.2.1. *Boundary cycles.*

$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^3a^0 \to {}^3b^0 \to {}^1b^2)$$
$$\gamma = ({}^2a^3 \to {}^0b^1)$$
$$\delta = ({}^2b^3)$$

2.2.2. *Automorphisms.*

A_0	a	b	0	1	2	3	α	β	γ	δ
$A_1{}^{\ddagger}$	b	a	2	3	0	1	δ	β	γ	α

2.2.3. *Markings.*

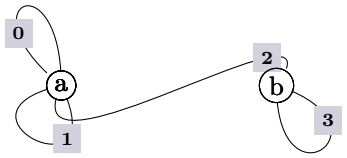
	$G_{4,1}^{(0)}$	$G_{4,1}^{(1)}$	$G_{4,1}^{(2)}$	$G_{4,1}^{(3)}$	$G_{4,1}^{(4)}$	$G_{4,1}^{(5)}$	$G_{4,1}^{(6)}$	$G_{4,1}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{4,1}^{(8)}$	$G_{4,1}^{(9)}$	$G_{4,1}^{(10)}$	$G_{4,1}^{(11)}$
α	1	1	2	2
β	2	3	0	1

(continued.)

γ	0	0	1	0
δ	3	2	3	3

2.3. The Fatgraph $G_{4,2}$ (24 orientable markings).



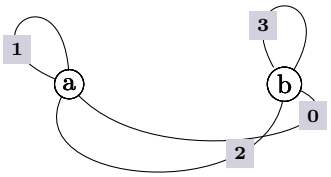

```
Fatgraph([
  Vertex([0, 0, 1, 2, 1]),# a
  Vertex([3, 3, 2]),      # b
])
```

2.3.1. *Boundary cycles.*

$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^4a^0)$$
$$\gamma = ({}^2a^3 \to {}^2b^0 \to {}^3a^4 \to {}^1b^2)$$
$$\delta = ({}^0b^1)$$

2.3.2. *Markings.* Fatgraph $G_{4,2}$ only has the identity automorphism, so the marked fatgraphs $G_{4,2}^{(0)}$ to $G_{4,2}^{(24)}$ are formed by decorating boundary cycles of $G_{4,2}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

2.4. The Fatgraph $G_{4,3}$ (non-orientable, 12 orientable markings).




```
Fatgraph([
  Vertex([1, 1, 2, 0]),# a
  Vertex([2, 0, 3, 3]),# b
])
```

2.4.1. *Boundary cycles.*

$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^3a^0 \to {}^0b^1)$$
$$\gamma = ({}^2a^3 \to {}^3b^0 \to {}^1b^2)$$
$$\delta = ({}^2b^3)$$

2.4.2. *Automorphisms.*

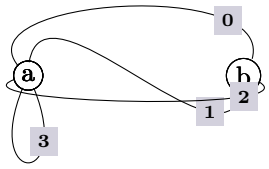
A_0	a	b	0	1	2	3	α	β	γ	δ
$A_1{}^{\dagger\ddagger}$	b	a	0	3	2	1	δ	γ	β	α

2.4.3. *Markings.*

	$G_{4,3}^{(0)}$	$G_{4,3}^{(1)}$	$G_{4,3}^{(2)}$	$G_{4,3}^{(3)}$	$G_{4,3}^{(4)}$	$G_{4,3}^{(5)}$	$G_{4,3}^{(6)}$	$G_{4,3}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{4,3}^{(8)}$	$G_{4,3}^{(9)}$	$G_{4,3}^{(10)}$	$G_{4,3}^{(11)}$
α	1	1	2	2
β	2	3	0	1
γ	0	0	1	0
δ	3	2	3	3

2.5. The Fatgraph $G_{4,4}$ (24 orientable markings).



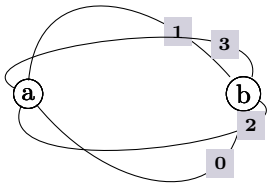

```
Fatgraph([
  Vertex([1, 0, 2, 3, 3]),# a
  Vertex([1, 2, 0]),      # b
])
```

2.5.1. *Boundary cycles.*

$$\alpha = ({}^2b^0 \to {}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^1b^2)$$
$$\gamma = ({}^2a^3 \to {}^0b^1 \to {}^4a^0)$$
$$\delta = ({}^3a^4)$$

2.5.2. *Markings.* Fatgraph $G_{4,4}$ only has the identity automorphism, so the marked fatgraphs $G_{4,4}^{(0)}$ to $G_{4,4}^{(24)}$ are formed by decorating boundary cycles of $G_{4,4}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

2.6. The Fatgraph $G_{4,5}$ (non-orientable, 3 orientable markings).




```
Fatgraph([
  Vertex([1, 3, 2, 0]),# a
  Vertex([0, 2, 3, 1]),# b
])
```

2.6.1. *Boundary cycles.*

$$\alpha = ({}^0a^1 \to {}^2b^3)$$
$$\beta = ({}^1a^2 \to {}^1b^2)$$
$$\gamma = ({}^2a^3 \to {}^0b^1)$$
$$\delta = ({}^3a^0 \to {}^3b^0)$$

2.6.2. *Automorphisms.*

A_0	a	b	0	1	2	3	α	β	γ	δ
$A_1{}^{\dagger\ddagger}$	a	b	1	3	0	2	β	γ	δ	α
$A_2{}^{\ddagger}$	a	b	3	2	1	0	γ	δ	α	β
$A_3{}^{\dagger\ddagger}$	a	b	2	0	3	1	δ	α	β	γ
$A_4{}^{\ddagger}$	b	a	1	0	3	2	γ	β	α	δ
$A_5{}^{\dagger\ddagger}$	b	a	0	2	1	3	β	α	δ	γ
$A_6{}^{\ddagger}$	b	a	2	3	0	1	α	δ	γ	β
$A_7{}^{\dagger\ddagger}$	b	a	3	1	2	0	δ	γ	β	α

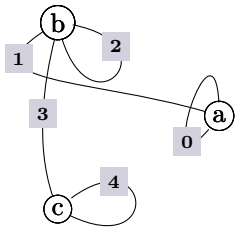
2.6.3. *Markings.*

	$G_{4,5}^{(0)}$	$G_{4,5}^{(1)}$	$G_{4,5}^{(2)}$
α	0	0	0
β	1	1	2
γ	2	3	1
δ	3	2	3

3. Fatgraphs with 5 edges / 3 vertices

There are 7 unmarked fatgraphs in this section, originating 288 marked fatgraphs (144 orientable, and 144 nonorientable).

3.1. The Fatgraph $G_{5,0}$ (24 orientable markings).




```
Fatgraph([
  Vertex([0, 1, 0]),   # a
  Vertex([1, 2, 2, 3]),# b
  Vertex([4, 4, 3]),   # c
])
```

3.1.1. *Boundary cycles.*

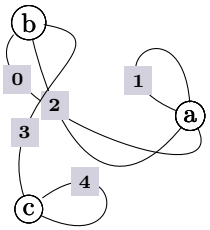
$$\alpha = ({}^1a^2 \to {}^0a^1 \to {}^2b^3 \to {}^3b^0 \to {}^1c^2 \to {}^2c^0 \to {}^0b^1)$$
$$\beta = ({}^2a^0)$$
$$\gamma = ({}^1b^2)$$
$$\delta = ({}^0c^1)$$

3.1.2. *Markings.* Fatgraph $G_{5,0}$ only has the identity automorphism, so the marked fatgraphs $G_{5,0}^{(0)}$ to $G_{5,0}^{(24)}$ are formed by decorating boundary cycles of $G_{5,0}$ with all permutations of $(0,1,2,3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

3.1.3. *Differentials.*

$$D(G_{5,0}^{(0)}) = +G_{4,0}^{(0)} \qquad D(G_{5,0}^{(11)}) = +G_{4,0}^{(4)}$$
$$D(G_{5,0}^{(1)}) = +G_{4,0}^{(1)} \qquad D(G_{5,0}^{(14)}) = +G_{4,0}^{(1)}$$
$$D(G_{5,0}^{(2)}) = +G_{4,0}^{(2)} \qquad D(G_{5,0}^{(15)}) = +G_{4,0}^{(0)}$$
$$D(G_{5,0}^{(3)}) = +G_{4,0}^{(3)} \qquad D(G_{5,0}^{(16)}) = +G_{4,0}^{(4)}$$
$$D(G_{5,0}^{(4)}) = +G_{4,0}^{(4)} \qquad D(G_{5,0}^{(17)}) = +G_{4,0}^{(5)}$$
$$D(G_{5,0}^{(5)}) = +G_{4,0}^{(5)} \qquad D(G_{5,0}^{(20)}) = +G_{4,0}^{(0)}$$
$$D(G_{5,0}^{(8)}) = +G_{4,0}^{(3)} \qquad D(G_{5,0}^{(21)}) = +G_{4,0}^{(1)}$$
$$D(G_{5,0}^{(9)}) = +G_{4,0}^{(2)} \qquad D(G_{5,0}^{(22)}) = +G_{4,0}^{(2)}$$
$$D(G_{5,0}^{(10)}) = +G_{4,0}^{(5)} \qquad D(G_{5,0}^{(23)}) = +G_{4,0}^{(3)}$$

3.2. The Fatgraph $G_{5,1}$ (24 orientable markings).




```
Fatgraph([
  Vertex([1, 1, 2, 0]),# a
  Vertex([0, 2, 3]),   # b
  Vertex([4, 4, 3]),   # c
])
```

3.2.1. *Boundary cycles.*

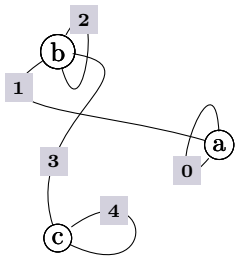
$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^2c^0 \to {}^3a^0 \to {}^1c^2 \to {}^2b^0 \to {}^1b^2)$$
$$\gamma = ({}^2a^3 \to {}^0b^1)$$
$$\delta = ({}^0c^1)$$

3.2.2. *Markings.* Fatgraph $G_{5,1}$ only has the identity automorphism, so the marked fatgraphs $G_{5,1}^{(0)}$ to $G_{5,1}^{(24)}$ are formed by decorating boundary cycles of $G_{5,1}$ with all permutations of $(0,1,2,3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

3.2.3. *Differentials.*

$$D(G_{5,1}^{(0)}) = +G_{4,0}^{(0)} + G_{4,0}^{(1)}$$
$$D(G_{5,1}^{(1)}) = +G_{4,0}^{(0)} + G_{4,0}^{(1)}$$
$$D(G_{5,1}^{(2)}) = +G_{4,0}^{(2)} + G_{4,0}^{(3)}$$
$$D(G_{5,1}^{(3)}) = +G_{4,0}^{(2)} + G_{4,0}^{(3)}$$
$$D(G_{5,1}^{(4)}) = +G_{4,0}^{(4)} + G_{4,0}^{(5)}$$
$$D(G_{5,1}^{(5)}) = +G_{4,0}^{(4)} + G_{4,0}^{(5)}$$
$$D(G_{5,1}^{(8)}) = +G_{4,0}^{(2)} + G_{4,0}^{(3)}$$
$$D(G_{5,1}^{(9)}) = +G_{4,0}^{(4)} + G_{4,0}^{(5)}$$
$$D(G_{5,1}^{(11)}) = +G_{4,0}^{(0)} + G_{4,0}^{(1)}$$

3.3. The Fatgraph $G_{5,2}$ (12 orientable markings).




```
Fatgraph([
  Vertex([0, 1, 0]),   # a
  Vertex([1, 2, 3, 2]),# b
  Vertex([4, 4, 3]),   # c
])
```

3.3.1. *Boundary cycles.*

$$\alpha = ({}^{1}a^{2} \to {}^{3}b^{0} \to {}^{0}a^{1} \to {}^{0}b^{1})$$
$$\beta = ({}^{2}a^{0})$$
$$\gamma = ({}^{2}b^{3} \to {}^{2}c^{0} \to {}^{1}c^{2} \to {}^{1}b^{2})$$
$$\delta = ({}^{0}c^{1})$$

3.3.2. *Automorphisms.*

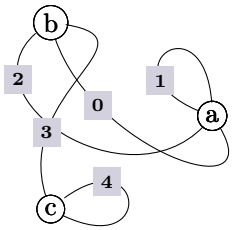
A_0	a	b	c	0	1	2	3	4	α	β	γ	δ
$A_1{}^{\ddagger}$	c	b	a	4	3	2	1	0	γ	δ	α	β

3.3.3. *Markings.*

	$G_{5,2}^{(0)}$	$G_{5,2}^{(1)}$	$G_{5,2}^{(2)}$	$G_{5,2}^{(3)}$	$G_{5,2}^{(4)}$	$G_{5,2}^{(5)}$	$G_{5,2}^{(6)}$	$G_{5,2}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{5,2}^{(8)}$	$G_{5,2}^{(9)}$	$G_{5,2}^{(10)}$	$G_{5,2}^{(11)}$
α	1	1	2	2
β	2	3	0	1
γ	3	2	3	3
δ	0	0	1	0

3.4. The Fatgraph $G_{5,3}$ (24 orientable markings).




```
Fatgraph([
  Vertex([1, 1, 2, 0]),# a
  Vertex([2, 0, 3]),   # b
  Vertex([4, 4, 3]),   # c
])
```

3.4.1. *Boundary cycles.*

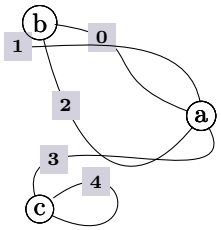
$$\alpha = ({}^0a^1)$$
$$\beta = ({}^1a^2 \to {}^3a^0 \to {}^0b^1)$$
$$\gamma = ({}^2a^3 \to {}^2b^0 \to {}^2c^0 \to {}^1c^2 \to {}^1b^2)$$
$$\delta = ({}^0c^1)$$

3.4.2. *Markings.* Fatgraph $G_{5,3}$ only has the identity automorphism, so the marked fatgraphs $G_{5,3}^{(0)}$ to $G_{5,3}^{(24)}$ are formed by decorating boundary cycles of $G_{5,3}$ with all permutations of $(0,1,2,3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

3.4.3. *Differentials.*

$$D(G_{5,3}^{(12)}) = +G_{4,0}^{(2)} \qquad D(G_{5,3}^{(17)}) = +G_{4,0}^{(3)}$$
$$D(G_{5,3}^{(13)}) = +G_{4,0}^{(4)} \qquad D(G_{5,3}^{(18)}) = +G_{4,0}^{(3)}$$
$$D(G_{5,3}^{(14)}) = +G_{4,0}^{(0)} \qquad D(G_{5,3}^{(19)}) = +G_{4,0}^{(5)}$$
$$D(G_{5,3}^{(15)}) = +G_{4,0}^{(5)} \qquad D(G_{5,3}^{(21)}) = +G_{4,0}^{(4)}$$
$$D(G_{5,3}^{(16)}) = +G_{4,0}^{(1)} \qquad D(G_{5,3}^{(23)}) = +G_{4,0}^{(2)}$$

3.5. The Fatgraph $G_{5,4}$ (24 orientable markings).




```
Fatgraph([
  Vertex([1, 0, 2, 3]),# a
  Vertex([1, 2, 0]),   # b
  Vertex([4, 4, 3]),   # c
])
```

3.5.1. *Boundary cycles.*

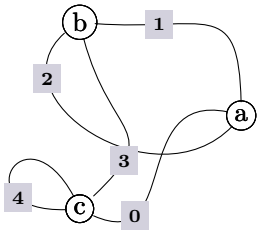
$$\alpha = ({}^{2}b^{0} \to {}^{0}a^{1})$$
$$\beta = ({}^{1}a^{2} \to {}^{1}b^{2})$$
$$\gamma = ({}^{2}a^{3} \to {}^{3}a^{0} \to {}^{0}b^{1} \to {}^{1}c^{2} \to {}^{2}c^{0})$$
$$\delta = ({}^{0}c^{1})$$

3.5.2. *Markings.* Fatgraph $G_{5,4}$ only has the identity automorphism, so the marked fatgraphs $G_{5,4}^{(0)}$ to $G_{5,4}^{(24)}$ are formed by decorating boundary cycles of $G_{5,4}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

3.5.3. *Differentials.*

$$D(G_{5,4}^{(0)}) = +G_{4,0}^{(1)} \qquad D(G_{5,4}^{(6)}) = +G_{4,0}^{(0)}$$
$$D(G_{5,4}^{(1)}) = +G_{4,0}^{(4)} \qquad D(G_{5,4}^{(7)}) = +G_{4,0}^{(2)}$$
$$D(G_{5,4}^{(3)}) = +G_{4,0}^{(5)} \qquad D(G_{5,4}^{(9)}) = +G_{4,0}^{(3)}$$
$$D(G_{5,4}^{(5)}) = +G_{4,0}^{(0)} \qquad D(G_{5,4}^{(11)}) = +G_{4,0}^{(1)}$$

3.6. The Fatgraph $G_{5,5}$ (24 orientable markings).



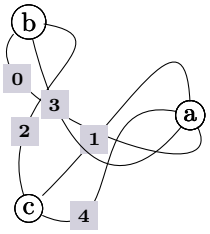

```
Fatgraph([
  Vertex([1, 0, 2]),   # a
  Vertex([2, 3, 1]),   # b
  Vertex([0, 3, 4, 4]),# c
])
```

3.6.1. *Boundary cycles.*

$$\alpha = ({}^0a^1 \to {}^0c^1 \to {}^1b^2)$$
$$\beta = ({}^1a^2 \to {}^3c^0 \to {}^1c^2 \to {}^0b^1)$$
$$\gamma = ({}^2a^0 \to {}^2b^0)$$
$$\delta = ({}^2c^3)$$

3.6.2. *Markings.* Fatgraph $G_{5,5}$ only has the identity automorphism, so the marked fatgraphs $G_{5,5}^{(0)}$ to $G_{5,5}^{(24)}$ are formed by decorating boundary cycles of $G_{5,5}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

3.7. The Fatgraph $G_{5,6}$ (12 orientable markings).




```
Fatgraph([
  Vertex([1, 4, 3, 0]),# a
  Vertex([0, 3, 2]),   # b
  Vertex([4, 1, 2]),   # c
])
```

3.7.1. *Boundary cycles.*

$$\alpha = ({}^0a^1 \to {}^0c^1)$$
$$\beta = ({}^1a^2 \to {}^2c^0 \to {}^1b^2)$$
$$\gamma = ({}^2a^3 \to {}^0b^1)$$
$$\delta = ({}^3a^0 \to {}^2b^0 \to {}^1c^2)$$

3.7.2. *Automorphisms.*

A_0	a	b	c	0	1	2	3	4	α	β	γ	δ
$A_1{}^{\ddagger}$	a	c	b	4	3	2	1	0	γ	δ	α	β

3.7.3. *Markings.*

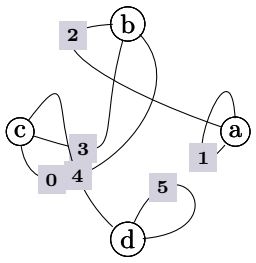
	$G_{5,6}^{(0)}$	$G_{5,6}^{(1)}$	$G_{5,6}^{(2)}$	$G_{5,6}^{(3)}$	$G_{5,6}^{(4)}$	$G_{5,6}^{(5)}$	$G_{5,6}^{(6)}$	$G_{5,6}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{5,6}^{(8)}$	$G_{5,6}^{(9)}$	$G_{5,6}^{(10)}$	$G_{5,6}^{(11)}$
α	1	1	2	2
β	2	3	0	1
γ	3	2	3	3
δ	0	0	1	0

4. Fatgraphs with 6 edges / 4 vertices

There are 6 unmarked fatgraphs in this section, originating 128 marked fatgraphs (64 orientable, and 64 nonorientable).

4.1. The Fatgraph $G_{6,0}$ (non-orientable, 12 orientable markings).




```
Fatgraph([
  Vertex([1, 2, 1]),# a
  Vertex([2, 3, 0]),# b
  Vertex([0, 3, 4]),# c
  Vertex([5, 5, 4]),# d
])
```

4.1.1. *Boundary cycles.*

$$\alpha = ({}^2d^0 \to {}^1a^2 \to {}^0a^1 \to {}^2c^0 \to {}^1c^2 \to {}^2b^0 \to {}^0b^1 \to {}^1d^2)$$
$$\beta = ({}^2a^0)$$
$$\gamma = ({}^0c^1 \to {}^1b^2)$$
$$\delta = ({}^0d^1)$$

4.1.2. *Automorphisms.*

A_0	a	b	c	d	0	1	2	3	4	5	α	β	γ	δ
$A_1{}^{\dagger\ddagger}$	d	c	b	a	3	5	4	0	2	1	α	δ	γ	β

4.1.3. *Markings.*

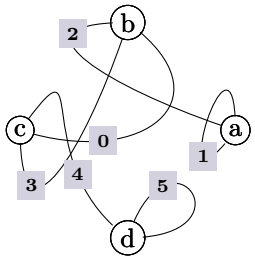
	$G_{6,0}^{(0)}$	$G_{6,0}^{(1)}$	$G_{6,0}^{(2)}$	$G_{6,0}^{(3)}$	$G_{6,0}^{(4)}$	$G_{6,0}^{(5)}$	$G_{6,0}^{(6)}$	$G_{6,0}^{(7)}$
α	0	0	0	1	1	1	2	2
β	1	1	2	0	0	2	0	0
γ	2	3	1	2	3	0	1	3
δ	3	2	3	3	2	3	3	1

	$G_{6,0}^{(8)}$	$G_{6,0}^{(9)}$	$G_{6,0}^{(10)}$	$G_{6,0}^{(11)}$
α	2	3	3	3
β	1	0	0	1
γ	0	1	2	0
δ	3	2	1	2

4.1.4. *Differentials.*

$$D(G_{6,0}^{(0)}) = -G_{5,0}^{(6)} \qquad D(G_{6,0}^{(7)}) = -G_{5,0}^{(1)}$$
$$D(G_{6,0}^{(4)}) = +G_{5,0}^{(6)} \qquad D(G_{6,0}^{(8)}) = +G_{5,0}^{(1)}$$
$$D(G_{6,0}^{(6)}) = -G_{5,0}^{(0)} \qquad D(G_{6,0}^{(10)}) = +G_{5,0}^{(0)}$$

4.2. The Fatgraph $G_{6,1}$ (12 orientable markings).




```
Fatgraph([
  Vertex([1, 2, 1]),# a
  Vertex([2, 3, 0]),# b
  Vertex([3, 0, 4]),# c
  Vertex([5, 5, 4]),# d
])
```

4.2.1. *Boundary cycles.*

$$\alpha = ({}^1a^2 \to {}^2b^0 \to {}^0a^1 \to {}^0c^1 \to {}^0b^1)$$
$$\beta = ({}^2a^0)$$
$$\gamma = ({}^2d^0 \to {}^2c^0 \to {}^1d^2 \to {}^1c^2 \to {}^1b^2)$$
$$\delta = ({}^0d^1)$$

4.2.2. *Automorphisms.*

A_0	a	b	c	d	0	1	2	3	4	5	α	β	γ	δ
$A_1{}^{\ddagger}$	d	c	b	a	0	5	4	3	2	1	γ	δ	α	β

4.2.3. *Markings.*

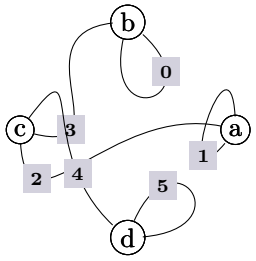
	$G_{6,1}^{(0)}$	$G_{6,1}^{(1)}$	$G_{6,1}^{(2)}$	$G_{6,1}^{(3)}$	$G_{6,1}^{(4)}$	$G_{6,1}^{(5)}$	$G_{6,1}^{(6)}$	$G_{6,1}^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2

	$G_{6,1}^{(8)}$	$G_{6,1}^{(9)}$	$G_{6,1}^{(10)}$	$G_{6,1}^{(11)}$
α	1	1	2	2
β	2	3	0	1
γ	3	2	3	3
δ	0	0	1	0

4.2.4. *Differentials.*

$$D(G_{6,1}^{(0)}) = -G_{5,0}^{(2)} \qquad D(G_{6,1}^{(6)}) = -G_{5,0}^{(4)}$$
$$D(G_{6,1}^{(1)}) = -G_{5,0}^{(3)} \qquad D(G_{6,1}^{(7)}) = -G_{5,0}^{(5)}$$
$$D(G_{6,1}^{(2)}) = +G_{5,0}^{(3)} \qquad D(G_{6,1}^{(8)}) = +G_{5,0}^{(5)}$$
$$D(G_{6,1}^{(4)}) = +G_{5,0}^{(2)} \qquad D(G_{6,1}^{(10)}) = +G_{5,0}^{(4)}$$

4.3. The Fatgraph $G_{6,2}$ (8 orientable markings).




```
Fatgraph([
  Vertex([1, 2, 1]),# a
  Vertex([3, 0, 0]),# b
  Vertex([2, 3, 4]),# c
  Vertex([5, 5, 4]),# d
])
```

4.3.1. *Boundary cycles.*

$\alpha = ({}^2d^0 \to {}^0c^1 \to {}^1a^2 \to {}^0a^1 \to {}^0b^1 \to {}^1c^2 \to {}^2b^0 \to {}^2c^0 \to {}^1d^2)$

$\beta = ({}^2a^0)$

$\gamma = ({}^1b^2)$

$\delta = ({}^0d^1)$

4.3.2. *Automorphisms.*

A_0	a	b	c	d	0	1	2	3	4	5	α	β	γ	δ
$A_1^{\ddagger}$	b	d	c	a	5	0	3	4	2	1	α	γ	δ	β
$A_2^{\ddagger}$	d	a	c	b	1	5	4	2	3	0	α	δ	β	γ

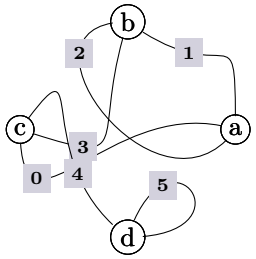
4.3.3. *Markings.*

	$G_{6,2}^{(0)}$	$G_{6,2}^{(1)}$	$G_{6,2}^{(2)}$	$G_{6,2}^{(3)}$	$G_{6,2}^{(4)}$	$G_{6,2}^{(5)}$	$G_{6,2}^{(6)}$	$G_{6,2}^{(7)}$
α	0	0	1	1	2	2	3	3
β	1	1	0	0	0	0	0	0
γ	2	3	2	3	1	3	1	2
δ	3	2	3	2	3	1	2	1

4.3.4. *Differentials.*

$$D(G^{(0)}_{6,2}) = +G^{(0)}_{5,0} \qquad D(G^{(4)}_{6,2}) = +G^{(4)}_{5,0}$$
$$D(G^{(1)}_{6,2}) = +G^{(1)}_{5,0}$$
$$D(G^{(2)}_{6,2}) = +G^{(2)}_{5,0} \qquad D(G^{(5)}_{6,2}) = +G^{(5)}_{5,0}$$
$$D(G^{(3)}_{6,2}) = +G^{(3)}_{5,0} \qquad D(G^{(6)}_{6,2}) = +G^{(6)}_{5,0}$$

4.4. The Fatgraph $G_{6,3}$ (24 orientable markings).




```
Fatgraph([
  Vertex([1, 0, 2]),# a
  Vertex([2, 3, 1]),# b
  Vertex([0, 3, 4]),# c
  Vertex([5, 5, 4]),# d
])
```

4.4.1. *Boundary cycles.*

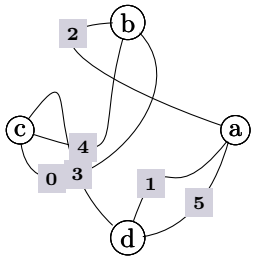
$$\alpha = ({}^0a^1 \to {}^0c^1 \to {}^1b^2)$$
$$\beta = ({}^2d^0 \to {}^1a^2 \to {}^2c^0 \to {}^1c^2 \to {}^0b^1 \to {}^1d^2)$$
$$\gamma = ({}^2a^0 \to {}^2b^0)$$
$$\delta = ({}^0d^1)$$

4.4.2. *Markings.* Fatgraph $G_{6,3}$ only has the identity automorphism, so the marked fatgraphs $G^{(0)}_{6,3}$ to $G^{(24)}_{6,3}$ are formed by decorating boundary cycles of $G_{6,3}$ with all permutations of $(0, 1, 2, 3)$ in lexicographic order. See Section 5 "Markings of fatgraphs with trivial automorphisms" for a complete table.

4.4.3. *Differentials.*

$$D(G^{(0)}_{6,3}) = +G^{(2)}_{5,0} \qquad D(G^{(8)}_{6,3}) = +G^{(5)}_{5,0}$$
$$D(G^{(2)}_{6,3}) = +G^{(4)}_{5,0}$$
$$D(G^{(4)}_{6,3}) = +G^{(6)}_{5,0} \qquad D(G^{(12)}_{6,3}) = +G^{(1)}_{5,0}$$
$$D(G^{(6)}_{6,3}) = +G^{(0)}_{5,0} \qquad D(G^{(14)}_{6,3}) = +G^{(3)}_{5,0}$$

4.5. The Fatgraph $G_{6,4}$ (non-orientable, 6 orientable markings).




```
Fatgraph([
  Vertex([5, 2, 1]),# a
  Vertex([2, 4, 0]),# b
  Vertex([0, 4, 3]),# c
  Vertex([5, 1, 3]),# d
])
```

4.5.1. *Boundary cycles.*

$$\alpha = ({}^{2}d^{0} \to {}^{0}a^{1} \to {}^{1}c^{2} \to {}^{0}b^{1})$$
$$\beta = ({}^{1}a^{2} \to {}^{2}b^{0} \to {}^{1}d^{2} \to {}^{2}c^{0})$$
$$\gamma = ({}^{2}a^{0} \to {}^{0}d^{1})$$
$$\delta = ({}^{0}c^{1} \to {}^{1}b^{2})$$

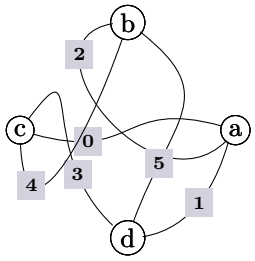
4.5.2. *Automorphisms.*

A_0	a	b	c	d	0	1	2	3	4	5	α	β	γ	δ
$A_1{}^{\ddagger}$	b	a	d	c	5	4	2	3	1	0	β	α	δ	γ
$A_2{}^{\dagger\ddagger}$	c	d	a	b	1	0	3	2	5	4	α	β	δ	γ
$A_3{}^{\dagger\ddagger}$	d	c	b	a	4	5	3	2	0	1	β	α	γ	δ

4.5.3. *Markings.*

	$G_{6,4}^{(0)}$	$G_{6,4}^{(1)}$	$G_{6,4}^{(2)}$	$G_{6,4}^{(3)}$	$G_{6,4}^{(4)}$	$G_{6,4}^{(5)}$
α	0	0	0	1	1	2
β	1	2	3	2	3	3
γ	2	1	1	0	0	0
δ	3	3	2	3	2	1

4.6. The Fatgraph $G_{6,5}$ (2 orientable markings).




```
Fatgraph([
  Vertex([1, 0, 2]),# a
  Vertex([2, 4, 5]),# b
  Vertex([4, 0, 3]),# c
  Vertex([1, 5, 3]),# d
])
```

4.6.1. *Boundary cycles.*

$$\alpha = ({}^2d^0 \to {}^0a^1 \to {}^1c^2)$$
$$\beta = ({}^1a^2 \to {}^0c^1 \to {}^0b^1)$$
$$\gamma = ({}^2a^0 \to {}^2b^0 \to {}^0d^1)$$
$$\delta = ({}^2c^0 \to {}^1d^2 \to {}^1b^2)$$

4.6.2. *Automorphisms.*

A_0	a	b	c	d	0	1	2	3	4	5	α	β	γ	δ
$A_1{}^\ddagger$	a	d	b	c	2	0	1	4	5	3	β	γ	α	δ
$A_2{}^\ddagger$	a	c	d	b	1	2	0	5	3	4	γ	α	β	δ
$A_3{}^\ddagger$	b	d	c	a	4	2	5	0	3	1	β	δ	γ	α
$A_4{}^\ddagger$	b	a	d	c	5	4	2	3	1	0	δ	γ	β	α
$A_5{}^\ddagger$	b	c	a	d	2	5	4	1	0	3	γ	β	δ	α
$A_6{}^\ddagger$	c	d	a	b	0	4	3	2	1	5	β	α	δ	γ
$A_7{}^\ddagger$	c	b	d	a	3	0	4	1	5	2	α	δ	β	γ
$A_8{}^\ddagger$	c	a	b	d	4	3	0	5	2	1	δ	β	α	γ
$A_9{}^\ddagger$	d	c	b	a	5	1	3	2	4	0	γ	δ	α	β
$A_{10}{}^\ddagger$	d	a	c	b	3	5	1	4	0	2	δ	α	γ	β
$A_{11}{}^\ddagger$	d	b	a	c	1	3	5	0	2	4	α	γ	δ	β

4.6.3. *Markings.*

	$G_{6,5}^{(0)}$	$G_{6,5}^{(1)}$
α	0	0
β	1	1
γ	2	3
δ	3	2

5. Markings of fatgraphs with trivial automorphisms

This section shows the numbering of marked fatgraphs when the base unmarked fatgraph G has only the trivial automorphism.

	$G^{(0)}$	$G^{(1)}$	$G^{(2)}$	$G^{(3)}$	$G^{(4)}$	$G^{(5)}$	$G^{(6)}$	$G^{(7)}$
α	0	0	0	0	0	0	1	1
β	1	1	2	2	3	3	0	0
γ	2	3	1	3	1	2	2	3
δ	3	2	3	1	2	1	3	2
	$G^{(8)}$	$G^{(9)}$	$G^{(10)}$	$G^{(11)}$	$G^{(12)}$	$G^{(13)}$	$G^{(14)}$	$G^{(15)}$
α	1	1	1	1	2	2	2	2
β	2	2	3	3	0	0	1	1
γ	0	3	0	2	1	3	0	3
δ	3	0	2	0	3	1	3	0
	$G^{(16)}$	$G^{(17)}$	$G^{(18)}$	$G^{(19)}$	$G^{(20)}$	$G^{(21)}$	$G^{(22)}$	$G^{(23)}$
α	2	2	3	3	3	3	3	3
β	3	3	0	0	1	1	2	2
γ	0	1	1	2	0	2	0	1
δ	1	0	2	1	2	0	1	0

APPENDIX D

Rheinfall benchmark data

The Sparse Integer Matrices Collection [15] is an online compilation of integer matrices from various sources, curated by Jean-Guillaume Dumas. It has been used for benchmarking the performance and other characteristics of the "Rheinfall" algorithm for Gaussian elimination (Chapter 4).

1. Features of the matrices in the SIMC collection

The following table recaps the main features of the matrices in the SIMC collection. The matrices are grouped according to the original subdivision in the SIMC website. Figures D.1 and D.2 provide a graphical depiction of the data in the table.

Matrix	Rows	Columns	Nonzero	Density%
BIBD				
BIBD_22_8_231x319770	231	319770	8953560	12.1212
bibd.11.5	55	462	4620	18.1818
bibd.12.4	66	495	2970	9.0909
bibd.15.3	105	455	1365	2.8571
bibd.17.3	136	680	2040	2.2059
bibd.9.3	36	84	252	8.3333
bibd.9.5	36	126	1260	27.7778
bibd_12_5_66x792	66	792	7920	15.1515
bibd_13_6_78x1716	78	1716	25740	19.2308
bibd_14_7_91x3432	91	3432	72072	23.0769
bibd_15_7_105x6435	105	6435	135135	20.0000
bibd_16_8_120x12870	120	12870	360360	23.3333
bibd_17_4_136x2380	136	2380	14280	4.4118
bibd_17_8_136x24310	136	24310	680680	20.5882
bibd_18_9_153x48620	153	48620	1750320	23.5294
bibd_19_9_171x92378	171	92378	3325608	21.0526
bibd_20_10_190x184756	190	184756	8314020	23.6842
bibd_49_3_1176x18424	1176	18424	55272	0.2551
bibd_81_2_3240x3240	3240	3240	3240	0.0309
bibd_81_3_3240x85320	3240	85320	255960	0.0926
CAG				
mat1916	1916	1916	195985	5.3387
mat364	364	364	13585	10.2531
mat72	72	72	1012	19.5216
Forest				
TF10	99	107	622	5.8718
TF11	216	236	1607	3.1525
TF12	488	552	4231	1.5707
TF13	1121	1302	11185	0.7663

Matrix	Rows	Columns	Nonzero	Density%
TF14	2644	3160	29862	0.3574
TF15	6334	7742	80057	0.1633
TF16	15437	19321	216173	0.0725
TF17	38132	48630	586218	0.0316
TF18	95368	123867	1597545	0.0135
TF19	241029	317955	4370721	5.703×10^{-03}
Franz				
10164x1740	10164	1740	40424	0.2286
1280x2800	1280	2800	11520	0.3214
16728x7176	16728	7176	100368	0.0836
19588x4164	19588	4164	97508	0.1195
19588x4164bis	19588	4164	97508	0.1195
47104x30144bis	47104	30144	329728	0.0232
6784x5252	6784	5252	46528	0.1306
7382x2882	7382	2882	44056	0.2071
7576x3016	7576	3016	45456	0.1989
big1sparse	2240	768	5120	0.2976
big2sparse	4032	4480	21504	0.1190
G5				
IG5-10	652	976	10273	1.6144
IG5-11	1227	1692	22110	1.0650
IG5-12	2296	2875	46260	0.7008
IG5-13	3994	4731	91209	0.4827
IG5-14	6735	7621	173337	0.3377
IG5-15	11369	11987	323509	0.2374
IG5-16	18846	18485	588326	0.1689
IG5-17	30162	27944	1035008	0.1228
IG5-18	47894	41550	1790490	0.0900
IG5-6	30	77	251	10.8658
IG5-7	62	150	549	5.9032
IG5-8	156	292	1711	3.7561
IG5-9	342	540	4570	2.4746
GL7d				
GL7d10	1	60	8	13.3333
GL7d11	1019	60	1513	2.4746
GL7d12	8899	1019	37519	0.4137
GL7d13	47271	8899	356232	0.0847
GL7d14	171375	47271	1831183	0.0226
GL7d15	460261	171375	6080381	7.709×10^{-03}
GL7d23	105054	349443	2695430	7.342×10^{-03}
GL7d24	21074	105054	593892	0.0268
GL7d25	2798	21074	81671	0.1385
GL7d26	305	2798	7412	0.8685
Grobner				
HFE18_96.in.sms	2372	4096	933343	9.6065
c8_mat11	4562	5761	2462970	9.3714
c8_mat11.I	4562	5761	2462970	9.3714
f855_mat9	2456	2511	171214	2.7763
f855_mat9.I	2456	2511	171214	2.7763
rkat7_mat5	694	738	38114	7.4416
robot24c1_mat5	404	302	15118	12.3910
robot24c1_mat5.J	302	404	15118	12.3910

Matrix	Rows	Columns	Nonzero	Density%
		Homology		
D6-6	120576	23740	147240	5.144×10^{-03}
ch3-3.b1.18x9	18	9	36	22.2222
ch3-3.b2.6x18	6	18	18	16.6667
ch4-4.b1.72x16	72	16	144	12.5000
ch4-4.b2.96x72	96	72	288	4.1667
ch4-4.b3.24x96	24	96	96	4.1667
ch5-5.b1.200x25	200	25	400	8.0000
ch5-5.b2.600x200	600	200	1800	1.5000
ch5-5.b3.600x600	600	600	2400	0.6667
ch5-5.b4.120x600	120	600	600	0.8333
ch6-6.b1.450x36	450	36	900	5.5556
ch6-6.b2.2400x450	2400	450	7200	0.6667
ch6-6.b3.5400x2400	5400	2400	21600	0.1667
ch6-6.b4.4320x5400	4320	5400	21600	0.0926
ch6-6.b5.720x4320	720	4320	4320	0.1389
ch7-6.b1.630x42	630	42	1260	4.7619
ch7-6.b2.4200x630	4200	630	12600	0.4762
ch7-6.b3.12600x4200	12600	4200	50400	0.0952
ch7-6.b4.15120x12600	15120	12600	75600	0.0397
ch7-6.b5.5040x15120	5040	15120	30240	0.0397
ch7-7.b1.882x49	882	49	1764	4.0816
ch7-7.b2.7350x882	7350	882	22050	0.3401
ch7-7.b5.35280x52920	35280	52920	211680	0.0113
ch7-8.b1.1176x56	1176	56	2352	3.5714
ch7-8.b2.11760x1176	11760	1176	35280	0.2551
ch7-8.b3.58800x11760	58800	11760	235200	0.0340
ch7-8.b4.141120x58800	141120	58800	705600	8.503×10^{-03}
ch7-8.b5.141120x141120	141120	141120	846720	4.252×10^{-03}
ch7-9.b1.1512x63	1512	63	3024	3.1746
ch7-9.b2.17640x1512	17640	1512	52920	0.1984
ch7-9.b3.105840x17640	105840	17640	423360	0.0227
ch7-9.b4.317520x105840	317520	105840	1587600	4.724×10^{-03}
ch7-9.b5.423360x317520	423360	317520	2540160	1.890×10^{-03}
ch8-8.b1.1568x64	1568	64	3136	3.1250
ch8-8.b2.18816x1568	18816	1568	56448	0.1913
ch8-8.b3.117600x18816	117600	18816	470400	0.0213
ch8-8.b4.376320x117600	376320	117600	1881600	4.252×10^{-03}
ch8-8.b5.564480x376320	564480	376320	3386880	1.594×10^{-03}
cis.n4c6.b1.210x21	210	21	420	9.5238
cis.n4c6.b13.6300x25605	6300	25605	88200	0.0547
cis.n4c6.b14.920x6300	920	6300	13800	0.2381
cis.n4c6.b15.60x920	60	920	960	1.7391
cis.n4c6.b2.1330x210	1330	210	3990	1.4286
cis.n4c6.b3.5970x1330	5970	1330	23880	0.3008
cis.n4c6.b4.20058x5970	20058	5970	100290	0.0838
klein.b1.30x10	30	10	60	20.0000
klein.b2.20x30	20	30	60	10.0000
lutz30-23.b6.1716x3003	1716	3003	12012	0.2331
m133.b3.200200x200200	200200	200200	800800	1.998×10^{-03}
mk10.b1.630x45	630	45	1260	4.4444
mk10.b2.3150x630	3150	630	9450	0.4762
mk10.b3.4725x3150	4725	3150	18900	0.1270
mk10.b4.945x4725	945	4725	4725	0.1058
mk11.b1.990x55	990	55	1980	3.6364

Matrix	Rows	Columns	Nonzero	Density%
mk11.b2.6930x990	6930	990	20790	0.3030
mk11.b3.17325x6930	17325	6930	69300	0.0577
mk11.b4.10395x17325	10395	17325	51975	0.0289
mk11.b4.9450x17325	9450	17325	47250	0.0289
mk12.b1.1485x66	1485	66	2970	3.0303
mk12.b2.13860x1485	13860	1485	41580	0.2020
mk12.b3.51975x13860	51975	13860	207900	0.0289
mk12.b4.62370x51975	62370	51975	311850	9.620×10^{-03}
mk12.b5.10395x62370	10395	62370	62370	9.620×10^{-03}
mk13.b5.135135x270270	135135	270270	810810	2.220×10^{-03}
mk9.b1.378x36	378	36	756	5.5556
mk9.b2.1260x378	1260	378	3780	0.7937
mk9.b3.945x1260	945	1260	3780	0.3175
n2c6.b1.105x15	105	15	210	13.3333
n2c6.b10.30x306	30	306	330	3.5948
n2c6.b2.455x105	455	105	1365	2.8571
n2c6.b3.1365x455	1365	455	5460	0.8791
n2c6.b4.3003x1365	3003	1365	15015	0.3663
n2c6.b5.4945x3003	4945	3003	29670	0.1998
n2c6.b6.5715x4945	5715	4945	40005	0.1416
n2c6.b7.3990x5715	3990	5715	31920	0.1400
n2c6.b8.1470x3990	1470	3990	13230	0.2256
n2c6.b9.306x1470	306	1470	3060	0.6803
n3c4.b1.15x6	15	6	30	33.3333
n3c4.b2.20x15	20	15	60	20.0000
n3c4.b3.15x20	15	20	60	20.0000
n3c4.b4.6x15	6	15	30	33.3333
n3c5.b1.45x10	45	10	90	20.0000
n3c5.b2.120x45	120	45	360	6.6667
n3c5.b3.210x120	210	120	840	3.3333
n3c5.b4.252x210	252	210	1260	2.3810
n3c5.b5.210x252	210	252	1260	2.3810
n3c5.b6.120x210	120	210	840	3.3333
n3c5.b7.30x120	30	120	240	6.6667
n3c6.b1.105x15	105	105	210	1.9048
n3c6.b10.675x2511	675	2511	7425	0.4381
n3c6.b11.60x675	60	675	720	1.7778
n3c6.b2.455x105	455	105	1365	2.8571
n3c6.b3.1365x455	1365	455	5460	0.8791
n3c6.b4.3003x1365	3003	1365	15015	0.3663
n3c6.b5.5005x3003	5005	3003	30030	0.1998
n3c6.b6.6435x5005	6435	5005	45045	0.1399
n3c6.b7.6435x6435	6435	6435	51480	0.1243
n3c6.b8.4935x6435	4935	6435	44415	0.1399
n3c6.b9.2511x4935	2511	4935	25110	0.2026
n4c5.b1.105x15	105	15	210	13.3333
n4c5.b10.120x630	120	630	1320	1.7460
n4c5.b11.10x120	10	120	120	10.0000
n4c5.b2.455x105	455	105	1365	2.8571
n4c5.b3.1350x455	1350	455	5400	0.8791
n4c5.b4.2852x1350	2852	1350	14260	0.3704
n4c5.b5.4340x2852	4340	2852	26040	0.2104
n4c5.b6.4735x4340	4735	4340	33145	0.1613
n4c5.b7.3635x4735	3635	4735	29080	0.1690
n4c5.b8.1895x3635	1895	3635	17055	0.2476

Matrix	Rows	Columns	Nonzero	Density%
n4c5.b9.630x1895	630	1895	6300	0.5277
n4c6.b1.210x21	210	21	420	9.5238
n4c6.b10.132402x186558	132402	186558	1456422	5.896×10^{-03}
n4c6.b11.69235x132402	69235	132402	830820	9.063×10^{-03}
n4c6.b12.25605x69235	25605	69235	332865	0.0188
n4c6.b13.6300x25605	6300	25605	88200	0.0547
n4c6.b14.920x6300	920	6300	13800	0.2381
n4c6.b15.60x920	60	920	960	1.7391
n4c6.b2.1330x210	1330	210	3990	1.4286
n4c6.b3.5970x1330	5970	1330	23880	0.3008
n4c6.b4.20058x5970	20058	5970	100290	0.0838
n4c6.b5.51813x20058	51813	20058	310878	0.0299
n4c6.b6.104115x51813	104115	51813	728805	0.0135
n4c6.b7.163215x104115	163215	104115	1305720	7.684×10^{-03}
n4c6.b8.198895x163215	198895	163215	1790055	5.514×10^{-03}
n4c6.b9.186558x198895	186558	198895	1865580	5.028×10^{-03}
shar_te2.b1.17160x286	17160	286	34320	0.6993
shar_te2.b2.200200x17160	200200	17160	600600	0.0175
shar_te2.b3.200200x200200	200200	200200	800800	1.998×10^{-03}
Kocay				
Trec10	106	478	8612	16.9969
Trec11	235	1138	35705	13.3512
Trec12	551	2726	151219	10.0677
Trec13	1301	6561	654517	7.6678
Trec14	3159	15905	2872265	5.7166
Trec4	2	3	3	50.0000
Trec5	3	7	12	57.1429
Trec6	6	15	40	44.4444
Trec7	11	36	147	37.1212
Trec8	23	84	549	28.4161
Trec9	47	201	2147	22.7268
Margulies				
cat_ears_2_1	85	85	254	3.5156
cat_ears_2_4	1009	2689	7982	0.2942
cat_ears_3_1	204	181	542	1.4679
cat_ears_3_4	5226	13271	39592	0.0571
cat_ears_4_1	377	313	938	0.7949
cat_ears_4_4	19020	44448	132888	0.0157
flower_4_1	121	129	386	2.4729
flower_4_4	1837	5529	16466	0.1621
flower_5_1	211	201	602	1.4194
flower_5_4	5226	14721	43942	0.0571
flower_7_1	463	393	1178	0.6474
flower_7_4	27693	67593	202218	0.0108
flower_8_1	625	513	1538	0.4797
flower_8_4	55081	125361	375266	5.435×10^{-03}
kneser_10_4_1	349651	330751	992252	8.580×10^{-04}
kneser_6_2_1	601	676	2027	0.4989
kneser_8_3_1	15737	15681	47042	0.0191
wheel_3_1	21	25	74	14.0952
wheel_4_1	36	41	122	8.2656
wheel_5_1	57	61	182	5.2344
wheel_601	902103	723605	2170814	3.326×10^{-04}
wheel_6_1	83	85	254	3.6003

Matrix	Rows	Columns	Nonzero	Density%
wheel_7_1	114	113	338	2.6238
		Relat		
rel3	12	5	18	30.0000
rel4	66	12	104	13.1313
rel5	340	35	656	5.5126
rel6	2340	157	5101	1.3885
rel7	21924	1045	50636	0.2210
rel8	345688	12347	821839	0.0193
relat3	12	5	24	40.0000
relat4	66	12	172	21.7172
relat5	340	35	1058	8.8908
relat6	2340	157	8108	2.2070
relat7	21924	1045	81355	0.3551
relat7b	21924	1045	81355	0.3551
relat8	345688	12347	1334038	0.0313
		Smooshed		
olivermatrix.1	9873	9621360	30002	3.158×10^{-05}
olivermatrix.2	78661	737004	1494559	2.578×10^{-03}
		Taha		
mat2_abtaha	37932	331	137228	1.0930
mat_abtaha	14596	209	51307	1.6819
		diffGL6		
D_10	163	341	2053	3.6936
D_6	469	201	2526	2.6796
D_7	636	470	5378	1.7991
D_8	544	637	6153	1.7756
D_9	340	545	4349	2.3470
		diffSL6		
D_10	460	816	7614	2.0285
D_11	169	461	2952	3.7890
D_5	434	115	1832	3.6706
D_6	970	435	6491	1.5383
D_7	1270	971	12714	1.0310
D_8	1132	1271	14966	1.0402
D_9	815	1133	12395	1.3423

2. Performance of different pivoting strategies

The following table lists the number of arithmetic operations performed by the "Rheinfall" algorithm with each of the different pivoting strategies implemented. Each column corresponds to a pivoting strategy as defined in Section 3.2; boldface marks the minimum in each row. Figure D.3 gives a graphical summary of the data in the table, color-coded by matrix group; a different view on the same data, grouped by pivoting algorithm, has been printed and analyzed in Figure 4.2 in Section 3.3.

The tests were run on the Swiss National Grid Infrastructure SMSCG [58] using the GC3Pie framework [17]. In contrast with timing measurements, the count of arithmetic operations only depends on the algorithm and is thus independent of

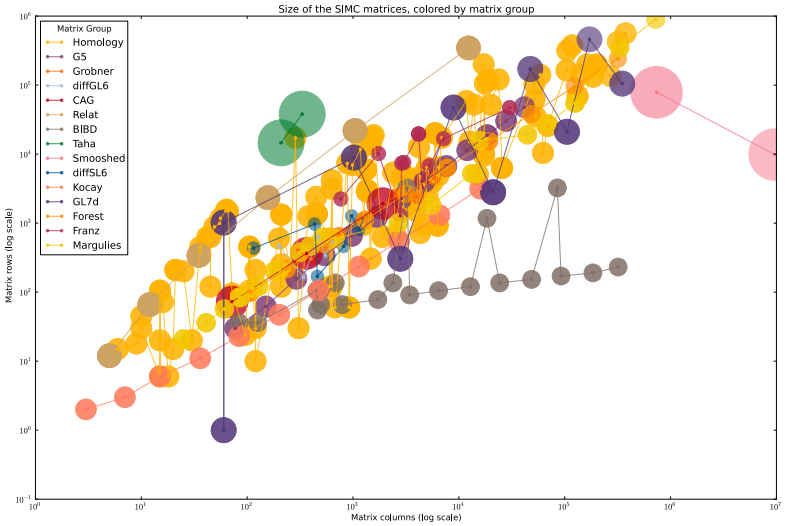


FIGURE D.1. Size and density of SIMC matrices, color-coded by group. Each matrix is represented by a dot, whose coordinates give the size of the matrix: the x-axis plots the number of columns (log scale), and the y-axis reports the number of rows (log scale). The radius of each colored disc is proportional to the logarithm of the matrix density.

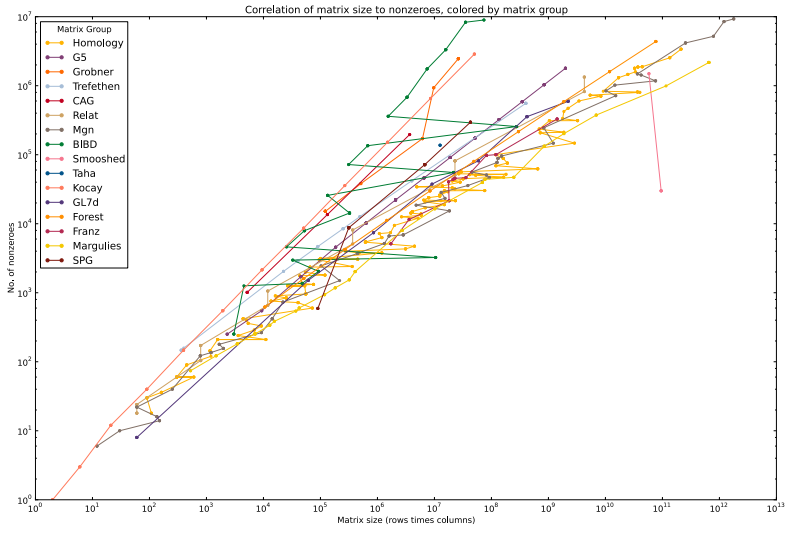


FIGURE D.2. Scatter plot of number of nonzero elements relative to matrix size. A colored thread joins matrices belonging to the same SIMC group. Note that both axes use log scale.

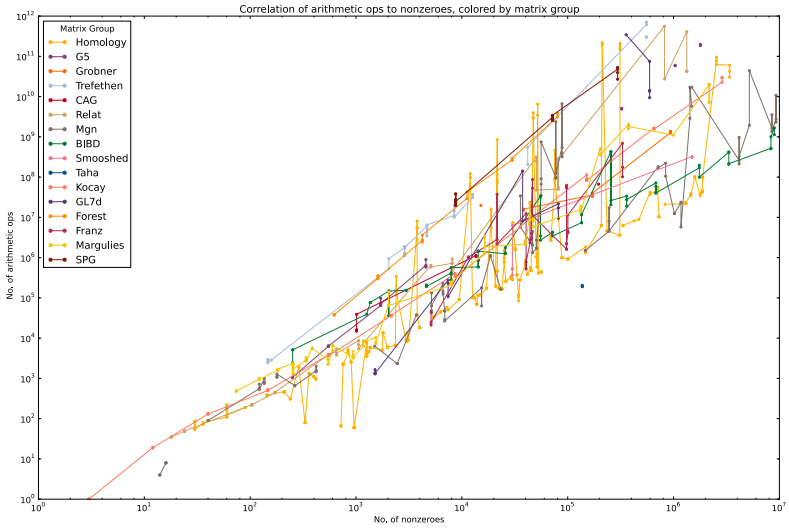


FIGURE D.3. Scatter plot of the number of arithmetic operations relative to number of nonzero elements, color-coded by matrix group. Note that both axes use log scale.

the hardware where the program is run. For each matrix, a computational job was spawned, which was allocated a maximum run time of 24 hours and 4GB maximum memory usage.

For some matrices it was not possible to obtain the precise count; these cases have been marked with one of the following labels:

Arith. ovf.:
: During the elimination work, a matrix entry exceeded the maximum integer number representable as a 64-bit machine integer.

No mem.:
: The rank-computation program run out of memory before completion.

No data:
: Some unknown error prevented the rank-computation program from completing; this includes the computational jobs that did not terminate within 24 hours.

Matrix	No pivoting	Sparsity	Threshold	Weight
		BIBD		
BIBD_22_8_231x319770	*No data*	*No data*	1.6623×10^{09}	1.1291×10^{09}
bibd.11.5	**191'030**	*No data*	209'937	*No data*
bibd.12.4	**153'849**	**153'849**	*No data*	*No data*
bibd.15.3	**76'968**	**76'968**	*No data*	*No data*
bibd.17.3	147'963	147'963	*No data*	**36'179**
bibd.17.4	1.4479×10^{06}	1.4479×10^{06}	855'426	**586'884**
bibd.9.3	116	116	*No data*	**469**
bibd.9.5	*No data*	*No data*	**39'294**	*No data*
bibd_12_5_66x792	384'962	416'165	564'994	**278'516**
bibd_13_6_78x1716	1.2677×10^{06}	1.2498×10^{06}	1.7685×10^{06}	1.3000×10^{06}
bibd_14_7_91x3432	3.4608×10^{06}	*No data*	*No data*	4.2818×10^{06}

Matrix	No pivoting	Sparsity	Threshold	Weight
bibd_15_7_105x6435	7.3755×10^{06}	7.4016×10^{06}	1.1690×10^{07}	*No data*
bibd_16_8_120x12870	1.8951×10^{07}	*No data*	3.3793×10^{07}	2.7751×10^{07}
bibd_17_4_136x2380	1.4479×10^{06}	*No data*	855'426	**586'884**
bibd_17_8_136x24310	4.0103×10^{07}	4.0685×10^{07}	7.1823×10^{07}	5.8319×10^{07}
bibd_18_9_153x48620	9.9416×10^{07}	*No data*	1.9710×10^{08}	1.7061×10^{08}
bibd_19_9_171x92378	2.0955×10^{08}	2.1453×10^{08}	4.1758×10^{08}	*No data*
bibd_20_10_190x184756	5.0389×10^{08}	5.2072×10^{08}	*No data*	1.0059×10^{09}
bibd_49_3_1176x18424	3.3598×10^{07}	3.3598×10^{07}	3.4545×10^{06}	2.7161×10^{06}
bibd_81_2_3240x3240	**0**	*No data*	**0**	**0**
bibd_81_3_3240x85320	4.2704×10^{08}	4.2704×10^{08}	2.6297×10^{07}	2.0469×10^{07}
CAG				
mat1916	*No mem.*	*No data*	6.6736×10^{07}	*No data*
mat364	*No data*	1.1223×10^{06}	1.0613×10^{06}	1.1039×10^{06}
mat72	39'011	**15'075**	16'197	15'166
Forest				
TF10	*No data*	37'607	**37'571**	38'378
TF11	354'926	**300'940**	308'425	310'284
TF12	3.6427×10^{06}	2.7682×10^{06}	2.6969×10^{06}	2.4411×10^{06}
TF13	4.2777×10^{07}	*No data*	2.9516×10^{07}	*No data*
TF14	*No data*	2.5990×10^{08}	2.9553×10^{08}	2.6800×10^{08}
TF15	*No data*	3.1290×10^{09}	3.8309×10^{09}	*No data*
TF16	*No mem.*	*No mem.*	*No mem.*	*No mem.*
TF17	*No data*	*No mem.*	*No mem.*	*No mem.*
TF18	*No data*	*No mem.*	*No mem.*	*No mem.*
TF19	*No mem.*	*No mem.*	*No mem.*	*No mem.*
Franz				
10164x1740	*No data*	1.2380×10^{07}	**523'688**	775'072
1280x2800	1.0291×10^{06}	1.0295×10^{06}	*No data*	*No data*
16728x7176	*No data*	1.1485×10^{07}	4.5349×10^{06}	4.1465×10^{06}
19588x4164	*No data*	6.2470×10^{07}	1.6203×10^{06}	2.2060×10^{06}
19588x4164bis	5.5493×10^{07}	5.4429×10^{07}	1.6223×10^{06}	2.2246×10^{06}
47104x30144bis	*No data*	6.9142×10^{08}	1.7558×10^{08}	1.0204×10^{08}
6784x5252	5.2780×10^{07}	8.6826×10^{07}	1.1200×10^{07}	*No data*
7382x2882	2.3057×10^{07}	*No data*	2.7179×10^{06}	2.3510×10^{06}
7576x3016	1.9337×10^{07}	1.8360×10^{07}	3.0515×10^{06}	4.0151×10^{06}
big1sparse	*No data*	44'522	**21'181**	24'698
big2sparse	3.7355×10^{07}	2.7931×10^{06}	*No data*	2.1845×10^{06}
G5				
IG5-10	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-11	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-12	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-13	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-14	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-15	*Arith. ovf.*	*Arith. ovf.*	4.9024×10^{09}	5.1063×10^{09}
IG5-16	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
IG5-17	*Arith. ovf.*	5.9988×10^{10}	*Arith. ovf.*	5.8572×10^{10}
IG5-18	*Arith. ovf.*	1.9806×10^{11}	1.8419×10^{11}	*Arith. ovf.*
IG5-6	*No data*	**058**	**058**	**058**
IG5-7	**269**	367	297	367
IG5-8	98'284	71'071	**64'371**	70'076
IG5-9	889'074	643'236	**583'278**	607'248

Matrix	No pivoting	Sparsity	Threshold	Weight
		GL7d		
GL7d10	**0**	**0**	**0**	**0**
GL7d11	653	333	**292**	339
GL7d12	1.4200×10^{08}	*No data*	8.1457×10^{06}	*No data*
GL7d13	3.4395×10^{11}	*No mem.*	*No mem.*	*No mem.*
GL7d16	*Arith. ovf.*	*No data*	*No data*	*No data*
GL7d19	*Arith. ovf.*	*No data*	*No data*	*No data*
GL7d20	*No data*	*No data*	*Arith. ovf.*	*Arith. ovf.*
GL7d24	7.5005×10^{10}	1.3539×10^{10}	9.4151×10^{09}	1.4280×10^{10}
GL7d25	*No data*	9.4973×10^{06}	2.1761×10^{07}	1.7159×10^{07}
GL7d26	275'805	**107'713**	*No data*	228'923
		Grobner		
HFE18_96.in.sms	1.3859×10^{09}	1.2632×10^{09}	*No data*	1.2596×10^{09}
c8_mat11	*No mem.*	*No data*	*No mem.*	*No mem.*
c8_mat11.I	*No mem.*	*No mem.*	*No mem.*	*No mem.*
f855_mat9	*No data*	3.3308×10^{07}	4.0252×10^{07}	3.6035×10^{07}
f855_mat9.I	3.3351×10^{07}	*No data*	4.0252×10^{07}	*No data*
rkat7_mat5	1.4933×10^{07}	*No data*	*No data*	1.6028×10^{07}
robot24c1_mat5	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*	*Arith. ovf.*
robot24c1_mat5.J	*No data*	*No mem.*	1.9902×10^{07}	*No mem.*
		Homology		
D6-6	1.8401×10^{06}	1.3160×10^{06}	*No data*	*No data*
ch3-3.b1.18x9	74	74	**72**	80
ch3-3.b2.6x18	**0**	**0**	**0**	*No data*
ch4-4.b1.72x16	388	388	**366**	382
ch4-4.b2.96x72	*No data*	008	**905**	930
ch5-5.b1.200x25	132	*No data*	**086**	114
ch5-5.b2.600x200	**13'596**	**13'596**	**13'596**	**13'596**
ch5-5.b3.600x600	346'025	159'318	**140'692**	*No data*
ch5-5.b4.120x600	**0**	**0**	**0**	*No data*
ch6-6.b1.450x36	570	570	**496**	*No data*
ch6-6.b2.2400x450	*No data*	**57'548**	**57'548**	*No data*
ch6-6.b3.5400x2400	*No data*	*No data*	**464'054**	464'614
ch6-6.b4.4320x5400	8.5019×10^{08}	8.7849×10^{07}	7.3530×10^{07}	1.1592×10^{08}
ch6-6.b5.720x4320	**0**	**0**	**0**	**0**
ch7-6.b1.630x42	599	599	**510**	563
ch7-6.b2.4200x630	*No data*	**101'158**	*No data*	**101'158**
ch7-6.b3.12600x4200	1.0515×10^{06}	*No data*	1.0486×10^{06}	1.0488×10^{06}
ch7-6.b4.15120x12600	4.6144×10^{08}	9.2025×10^{07}	8.0412×10^{07}	9.2349×10^{07}
ch7-6.b5.5040x15120	7.2952×10^{06}	7.2952×10^{06}	7.2952×10^{06}	*No data*
ch7-7.b1.882x49	026	026	*No data*	**984**
ch7-7.b2.7350x882	**175'380**	*No data*	*No data*	**175'380**
ch7-7.b5.35280x52920	*No data*	1.9189×10^{11}	*No data*	2.1726×10^{11}
ch7-8.b1.1176x56	687	687	**564**	639
ch7-8.b2.11760x1176	**278'622**	**278'622**	**278'622**	**278'622**
ch7-8.b3.58800x11760	4.5886×10^{06}	4.5886×10^{06}	4.5886×10^{06}	4.5886×10^{06}
ch7-8.b4.141120x58800	5.7063×10^{07}	5.1903×10^{07}	4.8583×10^{07}	4.8329×10^{07}
ch7-9.b1.1512x63	582	582	**442**	528
ch7-9.b2.17640x1512	**415'624**	*No data*	**415'624**	**415'624**
ch7-9.b3.105840x17640	8.1713×10^{06}	*No data*	*No data*	*No data*
ch7-9.b4.317520x105840	1.0218×10^{08}	1.0072×10^{08}	*No data*	*No data*
ch7-9.b5.423360x317520	*No data*	9.5007×10^{10}	6.2891×10^{10}	7.8621×10^{10}
ch8-8.b1.1568x64	884	884	*No data*	**830**

Matrix	No pivoting	Sparsity	Threshold	Weight
ch8-8.b2.18816x1568	**439'740**	**439'740**	**439'740**	**439'740**
ch8-8.b3.117600x18816	8.8644×10^{06}	*No data*	8.8644×10^{06}	*No data*
ch8-8.b4.376320x117600	9.8690×10^{07}	9.8690×10^{07}	9.7491×10^{07}	9.7513×10^{07}
ch8-8.b5.564480x376320	*No data*	5.9845×10^{10}	3.0472×10^{10}	4.2815×10^{10}
cis.n4c6.b1.210x21	**970**	*No data*	**970**	**970**
cis.n4c6.b13.6300x25605	**998'112**	*No data*	*No data*	1.0002×10^{06}
cis.n4c6.b14.920x6300	69'502	*No data*	**69'493**	*No data*
cis.n4c6.b15.60x920	**60**	**60**	**60**	**60**
cis.n4c6.b2.1330x210	**18'240**	**18'240**	*No data*	*No data*
cis.n4c6.b3.5970x1330	**164'220**	*No data*	**164'220**	**164'220**
cis.n4c6.b4.20058x5970	**918'830**	*No data*	**918'830**	**918'830**
klein.b1.30x10	121	*No data*	**115**	121
klein.b2.20x30	**110**	**110**	*No data*	**110**
lutz30-23.b6.1716x3003	1.2249×10^{08}	9.7790×10^{07}	*No data*	*No data*
m133.b3.200200x200200	*No mem.*	*No data*	*No data*	*No mem.*
mk10.b1.630x45	826	826	**626**	*No data*
mk10.b2.3150x630	**91'152**	*No data*	**91'152**	*No data*
mk10.b3.4725x3150	1.5986×10^{07}	8.5148×10^{06}	5.6036×10^{06}	*No data*
mk10.b4.945x4725	**0**	**0**	*No data*	**0**
mk11.b1.990x55	*No data*	970	**718**	*No data*
mk11.b2.6930x990	*No data*	**197'124**	**197'124**	**197'124**
mk11.b3.17325x6930	3.3754×10^{06}	3.0171×10^{06}	2.6560×10^{06}	2.6967×10^{06}
mk11.b4.10395x17325	3.4682×10^{09}	6.5338×10^{09}	*No data*	*No data*
mk11.b4.9450x17325	3.9372×10^{09}	2.9856×10^{09}	*No data*	2.8960×10^{09}
mk12.b1.1485x66	*No data*	*No data*	**577**	765
mk12.b2.13860x1485	**384'784**	*No data*	*No data*	**384'784**
mk12.b3.51975x13860	6.3085×10^{06}	6.3085×10^{06}	6.1795×10^{06}	6.1816×10^{06}
mk12.b4.62370x51975	*No mem.*	2.0674×10^{11}	1.4935×10^{11}	1.5892×10^{11}
mk12.b5.10395x62370	**0**	**0**	**0**	**0**
mk13.b5.135135x270270	*No mem.*	*No mem.*	*No mem.*	*No mem.*
mk9.b1.378x36	306	306	**154**	240
mk9.b2.1260x378	**37'010**	**37'010**	**37'010**	*No data*
mk9.b3.945x1260	8.1207×10^{06}	5.2889×10^{06}	2.8486×10^{06}	5.5121×10^{06}
n2c6.b1.105x15	**469**	*No data*	**469**	**469**
n2c6.b10.30x306	**80**	**80**	**80**	**80**
n2c6.b2.455x105	**824**	**824**	**824**	*No data*
n2c6.b3.1365x455	**34'034**	*No data*	*No data*	**34'034**
n2c6.b4.3003x1365	**120'120**	**120'120**	*No data*	**120'120**
n2c6.b5.4945x3003	297'089	296'949	*No data*	**294'997**
n2c6.b6.5715x4945	878'780	*No data*	**646'170**	708'290
n2c6.b7.3990x5715	1.7394×10^{06}	*No data*	1.5878×10^{06}	1.5430×10^{06}
n2c6.b8.1470x3990	100'366	100'141	**97'926**	*No data*
n2c6.b9.306x1470	**13'182**	**13'182**	**13'182**	**13'182**
n3c4.b1.15x6	**55**	**55**	*No data*	**55**
n3c4.b2.20x15	**184**	**184**	**184**	*No data*
n3c4.b3.15x20	**215**	**215**	**215**	**215**
n3c4.b4.6x15	**86**	**86**	*No data*	*No data*
n3c5.b1.45x10	**189**	**189**	**189**	**189**
n3c5.b2.120x45	**344**	**344**	*No data*	**344**
n3c5.b3.210x120	**284**	**284**	**284**	**284**
n3c5.b4.252x210	**560**	**560**	**560**	**560**
n3c5.b5.210x252	**980**	*No data*	*No data*	*No data*
n3c5.b6.120x210	**040**	**040**	**040**	**040**
n3c5.b7.30x120	**309**	*No data*	**309**	**309**
n3c6.b1.105x15	**455**	**455**	**455**	*No data*

Matrix	No pivoting	Sparsity	Threshold	Weight
n3c6.b10.675x2511	51'737	*No data*	*No data*	**50'216**
n3c6.b11.60x675	**66**	**66**	**66**	**66**
n3c6.b2.455x105	**824**	**824**	*No data*	*No data*
n3c6.b3.1365x455	**34'034**	**34'034**	*No data*	*No data*
n3c6.b4.3003x1365	**120'120**	**120'120**	*No data*	*No data*
n3c6.b5.5005x3003	**285'285**	**285'285**	**285'285**	*No data*
n3c6.b6.6435x5005	**480'480**	**480'480**	**480'480**	**480'480**
n3c6.b7.6435x6435	*No data*	**588'588**	**588'588**	**588'588**
n3c6.b8.4935x6435	*No data*	*No data*	**546'525**	568'035
n3c6.b9.2511x4935	1.5025×10^{06}	989'665	**770'889**	777'099
n4c5.b1.105x15	*No data*	*No data*	**469**	**469**
n4c5.b10.120x630	**596**	**596**	**596**	**596**
n4c5.b11.10x120	*No data*	*No data*	**0**	*No data*
n4c5.b2.455x105	*No data*	**824**	**824**	*No data*
n4c5.b3.1350x455	**33'772**	**33'772**	**33'772**	*No data*
n4c5.b4.2852x1350	**115'932**	**115'932**	**115'932**	**115'932**
n4c5.b5.4340x2852	260'156	260'156	*No data*	**255'521**
n4c5.b6.4735x4340	380'009	379'582	*No data*	**365'542**
n4c5.b7.3635x4735	*No data*	*No data*	334'518	**328'146**
n4c5.b8.1895x3635	176'637	*No data*	174'944	**169'601**
n4c5.b9.630x1895	44'936	44'936	45'014	**43'968**
n4c6.b1.210x21	**970**	**970**	**970**	*No data*
n4c6.b10.132402x186558	*No data*	3.9015×10^{07}	3.5139×10^{07}	3.7091×10^{07}
n4c6.b11.69235x132402	*No data*	2.1085×10^{07}	*No data*	*No data*
n4c6.b12.25605x69235	*No data*	6.3680×10^{06}	6.2191×10^{06}	6.2240×10^{06}
n4c6.b13.6300x25605	**998'112**	**998'112**	1.0090×10^{06}	*No data*
n4c6.b14.920x6300	69'502	69'502	**69'493**	**69'493**
n4c6.b15.60x920	**60**	**60**	**60**	**60**
n4c6.b2.1330x210	**18'240**	**18'240**	**18'240**	**18'240**
n4c6.b3.5970x1330	**164'220**	*No data*	**164'220**	**164'220**
n4c6.b4.20058x5970	**918'830**	*No data*	**918'830**	*No data*
n4c6.b5.51813x20058	3.6217×10^{06}	3.6215×10^{06}	3.5710×10^{06}	*No data*
n4c6.b6.104115x51813	*No data*	1.0565×10^{07}	1.0182×10^{07}	1.0310×10^{07}
n4c6.b7.163215x104115	2.3279×10^{07}	2.3279×10^{07}	2.1753×10^{07}	2.2337×10^{07}
n4c6.b8.198895x163215	3.8373×10^{07}	3.8234×10^{07}	3.4786×10^{07}	3.6373×10^{07}
n4c6.b9.186558x198895	4.7002×10^{07}	4.6349×10^{07}	4.1629×10^{07}	4.3913×10^{07}
shar_te2.b1.17160x286	119'325	119'325	**85'060**	104'928
shar_te2.b2.200200x17160	4.2521×10^{07}	3.7335×10^{07}	*No data*	*No data*
Kocay				
Trec10	399'325	402'318	**332'659**	398'486
Trec11	5.9528×10^{06}	5.7429×10^{06}	*No data*	*No data*
Trec12	1.1322×10^{08}	9.1119×10^{07}	8.2281×10^{07}	8.3029×10^{07}
Trec13	1.6610×10^{09}	*No data*	1.5590×10^{09}	*No data*
Trec14	2.2841×10^{10}	2.9927×10^{10}	2.9209×10^{10}	*No data*
Trec3	**0**	**0**	**0**	**0**
Trec4	**1**	*No data*	**1**	**1**
Trec5	*No data*	**19**	**19**	**19**
Trec6	**130**	136	**130**	*No data*
Trec7	532	**505**	**505**	**505**
Trec8	**627**	945	928	813
Trec9	**35'351**	*No data*	36'896	36'103
Margulies				
cat_ears_2_1	706	344	**310**	330
cat_ears_2_4	247'722	221'138	221'668	**208'555**

Matrix	No pivoting	Sparsity	Threshold	Weight
cat_ears_3_1	000	*No data*	*No data*	**215**
cat_ears_3_4	2.5961×10^{06}	2.1563×10^{06}	2.2029×10^{06}	2.1660×10^{06}
cat_ears_4_1	612	383	**181**	369
cat_ears_4_4	1.9027×10^{07}	1.4150×10^{07}	1.5787×10^{07}	1.4054×10^{07}
flower_4_1	729	*No data*	**495**	*No data*
flower_4_4	3.3117×10^{06}	1.8387×10^{06}	1.3120×10^{06}	*No data*
flower_5_1	673	**912**	553	*No data*
flower_5_4	2.3196×10^{07}	1.1545×10^{07}	8.6861×10^{06}	1.0870×10^{07}
flower_7_1	141	*No data*	603	**348**
flower_7_4	*No data*	5.0832×10^{08}	3.5229×10^{08}	*No data*
flower_8_1	*No data*	**507**	10'301	548
flower_8_4	*No data*	2.0598×10^{09}	1.5801×10^{09}	1.8146×10^{09}
kneser_10_4_1	*No data*	*No data*	1.1018×10^{09}	*No data*
kneser_6_2_1	141'909	65'058	64'532	**64'192**
kneser_8_3_1	*No data*	5.8170×10^{06}	8.0534×10^{06}	9.4021×10^{06}
wheel_3_1	*No data*	*No data*	496	**471**
wheel_4_1	004	**940**	**940**	*No data*
wheel_5_1	674	600	**597**	617
wheel_601	7.3159×10^{09}	1.9754×10^{10}	7.1284×10^{09}	1.9844×10^{10}
wheel_6_1	413	128	**048**	191
wheel_7_1	334	828	**683**	946
	Mgn			
M0,3-D2	**0**	**0**	**0**	*No data*
M0,4-D3	**0**	**0**	*No data*	**0**
M0,4-D4	**452**	549	012	549
M0,4-D5	663	*No data*	**653**	*No data*
M0,5-D4	**0**	*No data*	**0**	**0**
M0,5-D5	*No data*	**164'630**	*No data*	**164'630**
M0,5-D6	*No data*	2.6830×10^{06}	1.7126×10^{06}	1.7049×10^{06}
M0,5-D7	4.4588×10^{06}	2.0491×10^{06}	*No data*	1.3293×10^{06}
M0,5-D8	176'792	176'792	**63'027**	*No data*
M0,6-D10	*No data*	4.4132×10^{10}	*No data*	1.9348×10^{09}
M0,6-D11	2.3607×10^{07}	2.3607×10^{07}	5.7557×10^{06}	*No data*
M0,6-D5	**0**	**0**	**0**	**0**
M0,6-D6	1.2495×10^{07}	*No data*	1.2495×10^{07}	*No data*
M0,6-D7	9.6351×10^{08}	3.2719×10^{08}	2.1239×10^{08}	2.5750×10^{08}
M0,6-D8	*No data*	3.5480×10^{09}	1.3787×10^{09}	1.6375×10^{09}
M0,6-D9	*No data*	1.0844×10^{10}	2.3271×10^{09}	2.7290×10^{09}
M1,2-D3	**0**	*No data*	*No data*	**0**
M1,2-D4	**4**	*No data*	**4**	**4**
M1,2-D5	**8**	**8**	**8**	**8**
M1,3-D4	**0**	*No data*	**0**	**0**
M1,3-D5	**37'892**	*No data*	*No data*	*No data*
M1,3-D6	228'656	143'605	*No data*	**124'049**
M1,3-D7	132'080	63'092	**42'576**	48'560
M1,3-D8	**320**	*No data*	*No data*	*No data*
M1,4-D10	*No data*	*No data*	1.7063×10^{08}	1.8067×10^{08}
M1,4-D11	1.5083×10^{06}	1.5083×10^{06}	*No data*	*No data*
M1,4-D5	*No data*	**0**	**0**	*No data*
M1,4-D6	1.7332×10^{07}	8.0099×10^{06}	4.5990×10^{06}	4.5662×10^{06}
M1,4-D7	*No data*	2.2243×10^{08}	*No data*	1.0557×10^{08}
M1,4-D8	*Arith. ovf.*	1.7117×10^{10}	*No data*	5.7799×10^{09}
M1,4-D9	*No data*	1.7035×10^{10}	*No data*	2.8930×10^{09}
M2,1-D4	**0**	**0**	**0**	*No data*
M2,1-D5	722	557	**534**	557

Matrix	No pivoting	Sparsity	Threshold	Weight
M2,1-D6	286	**063**	*No data*	114
M2,1-D7	**755**	768	989	808
M2,1-D8	**90**	**90**	*No data*	*No data*
M2,2-D10	3.4248×10^{07}	5.7100×10^{06}	*No data*	*No data*
M2,2-D11	47'487	47'487	**26'666**	29'019
M2,2-D5	*No data*	**346**	**346**	*No data*
M2,2-D6	*No data*	1.0674×10^{06}	1.1323×10^{06}	*No data*
M2,2-D7	7.4924×10^{08}	9.0328×10^{07}	3.4661×10^{07}	6.7032×10^{07}
M2,2-D8	6.6522×10^{09}	5.4543×10^{08}	3.1860×10^{08}	3.9807×10^{08}
M2,2-D9	2.3774×10^{09}	1.2261×10^{08}	9.7906×10^{07}	9.4398×10^{07}
Relat				
rel3	**35**	**35**	**35**	**35**
rel4	224	224	224	**215**
rel5	230	**775**	335	*No data*
rel6	645'129	**583'916**	*No data*	*No data*
rel7	3.1401×10^{08}	2.4799×10^{08}	4.7687×10^{07}	*No data*
rel8	*No data*	5.5958×10^{11}	2.7419×10^{10}	*No data*
rel9	*No data*	*Arith. ovf.*	*No data*	*No data*
relat3	**49**	**49**	**49**	**49**
relat4	448	448	457	**439**
relat5	*No data*	888	650	**486**
relat6	923'954	724'414	**460'639**	*No data*
relat7	2.9076×10^{08}	2.6891×10^{08}	*No data*	5.0470×10^{07}
relat7b	*No data*	2.6891×10^{08}	5.5258×10^{07}	5.0470×10^{07}
relat8	*No data*	4.1028×10^{11}	4.2837×10^{10}	4.1730×10^{10}
relat9	*No data*	*No data*	*No data*	*Arith. ovf.*
SPG				
08blocks	**0**	**0**	**0**	*No data*
EX1	3.8339×10^{07}	2.5907×10^{07}	1.9590×10^{07}	2.1236×10^{07}
EX2	3.7665×10^{07}	2.8047×10^{07}	2.3384×10^{07}	2.4631×10^{07}
EX3	2.5611×10^{09}	3.0573×10^{09}	3.4014×10^{09}	2.6785×10^{09}
EX4	2.8603×10^{09}	2.9172×10^{09}	3.4216×10^{09}	2.5813×10^{09}
EX5	2.7025×10^{10}	*No data*	*No data*	4.7854×10^{10}
EX6	*No data*	5.3271×10^{10}	*No mem.*	3.9488×10^{10}
Smooshed				
olivermatrix.1	6.2721×10^{06}	533'298	**368'365**	518'246
olivermatrix.2	3.2262×10^{08}	*No data*	*No data*	3.0554×10^{08}
Taha				
mat2_abtaha	204'728	192'744	**190'097**	*No data*
Trefethen				
trefethen_150	938'010	*No data*	**527'183**	*No data*
trefethen_20	*No data*	*No data*	**858**	*No data*
trefethen_200	1.8745×10^{06}	1.7976×10^{06}	1.1273×10^{06}	1.2708×10^{06}
trefethen_2000	*No mem.*	*No mem.*	5.4902×10^{08}	2.0612×10^{08}
trefethen_20000	3.0268×10^{11}	7.0883×10^{11}	*No mem.*	*No mem.*
trefethen_20000__19999_minor	3.0078×10^{11}	6.1316×10^{11}	*No mem.*	*No mem.*
trefethen_200__199_minor	1.8429×10^{06}	1.7754×10^{06}	1.1140×10^{06}	1.2480×10^{06}
trefethen_20__19_minor	942	930	**453**	581
trefethen_300	6.4986×10^{06}	6.0418×10^{06}	*No data*	3.4885×10^{06}
trefethen_500	1.4487×10^{07}	2.0348×10^{07}	1.1111×10^{07}	1.0143×10^{07}
trefethen_700	*No data*	*No data*	3.7381×10^{07}	3.1335×10^{07}

Matrix	No pivoting	Sparsity	Threshold	Weight

3. Performance of different algorithms

The following table presents a comparison of the running times of the "Rheinfall" algorithm and the three rank-computing algorithms provided by LinBox: "Black Box" [63, 38], Sparse Elimination with Linear Pivoting (a GEPP implementation using Markowitz pivoting [13]), and Sparse Elimination with No Pivoting.

The "Rheinfall" column reports execution time (in seconds) for the sample C++ implementation. The "LinBox" columns report times for the GEPP implementation in LinBox version 1.1.7. Boldface font marks the best performer in each row.

The programs were run on the University of Zurich "Schroedinger" cluster, equipped with Intel Xeon X5560 CPUs @ 2.8GHz and running 64-bit SLES 11.1 Linux; codes were compiled with GCC 4.5.0 using options `-O3 -march=native`. Each computational job was allowed to use 24GB of memory maximum, and to run for 48 hours.

The system timer has a resolution of 4ms,[1] but scheduling jitter makes the measurements less accurate than that. However, we have not attempted an exact measurements of the uncertainty, as the purpose of the test was to determine the relative speed of the different algorithms and the performance differences are quite evident for most matrices.

For some matrices it was not possible to obtain the precise count; these cases have been marked with one of the following labels:

Arith. ovf.:
: During the elimination work, a matrix entry exceeded the maximum integer number representable as a 64-bit machine integer.

No mem.:
: The rank-computation program run out of memory before completion.

***No data*, *Error*:**
: Some unknown error prevented the rank-computation program from completing; this includes the computational jobs that did not terminate within the allotted time.

Matrix	LinBox			Rheinfall
	Black Box	Linear Pvt.	No Pvt.	Sparsity Pvt.
BIBD				
BIBD_22_8_231x319770	11.670	12.560	7.660	**5.428**
bibd_13_6_78x1716	0.010	0.010	0.020	**0.004**
bibd_14_7_91x3432	0.040	0.030	0.020	**0.008**
bibd_15_7_105x6435	0.080	0.080	0.060	**0.020**
bibd_16_8_120x12870	0.240	0.290	0.160	**0.056**
bibd_17_4_136x2380	0.010	0.010	0.020	**0.004**
bibd_17_8_136x24310	0.500	0.610	0.320	**0.164**
bibd_18_9_153x48620	1.320	1.480	0.910	**0.456**
bibd_19_9_171x92378	3.060	3.540	2.040	**1.272**
bibd_20_10_190x184756	8.260	10.560	4.990	**2.284**
bibd_49_3_1176x18424	45.770	**0.050**	0.420	0.096

[1] See the Linux man page `times(7)` for an explanation.

Matrix	LinBox			Rheinfall
	Black Box	Linear Pvt.	No Pvt.	Sparsity Pvt.
bibd_81_2_3240x3240	42.310	0.040	0.030	**<4ms**
bibd_81_3_3240x85320	1046.690	**0.530**	7.650	1.264
CAG				
mat1916	16.980	**0.300**	3.060	*No mem.*
mat364	0.010	**<4ms**	0.030	0.008
Forest				
TF11	**<4ms**	**<4ms**	0.010	**<4ms**
TF12	0.030	0.030	0.050	**0.012**
TF13	0.840	0.490	0.600	**0.064**
TF14	5.360	7.990	7.760	**0.820**
TF15	42.840	117.410	97.850	**17.505**
TF16	**377.590**	1884.560	1276.740	*No mem.*
TF17	**1932.390**	*No data*	17056.550	*No mem.*
TF18	**14392.000**	*No data*	*No data*	*No mem.*
TF19	**95106.390**	*No data*	*No data*	*No mem.*
Franz				
10164x1740	5.340	0.200	2.630	**0.164**
1280x2800	1.780	0.020	0.050	**0.008**
16728x7176	74.250	3.670	49.280	**0.156**
19588x4164	49.010	1.500	33.090	**0.720**
19588x4164bis	42.550	1.310	34.630	**0.660**
47104x30144bis	1120.790	69.800	3854.450	**4.180**
6784x5252	14.870	**0.460**	5.180	0.552
7382x2882	8.440	0.440	4.950	**0.076**
7576x3016	9.540	0.480	9.800	**0.172**
big1sparse	0.020	0.020	0.020	**0.004**
big2sparse	6.430	0.190	0.450	**0.020**
G5				
IG5-10	**0.020**	**0.020**	0.100	*Arith. ovf.*
IG5-11	1.600	**0.040**	0.560	*Arith. ovf.*
IG5-12	5.580	**0.120**	3.930	*Arith. ovf.*
IG5-13	19.650	**0.380**	21.020	*Arith. ovf.*
IG5-14	69.260	**1.440**	81.950	*Arith. ovf.*
IG5-15	196.120	**3.720**	295.060	*Arith. ovf.*
IG5-16	537.300	**9.320**	1283.360	*Arith. ovf.*
IG5-17	1414.290	**21.970**	4572.660	375.167
IG5-18	3646.680	**50.000**	*No data*	1191.666
IG5-9	**<4ms**	**<4ms**	0.010	0.004
GL7d				
GL7d12	2.870	0.130	0.160	**0.072**
GL7d13	304.340	**19.500**	521.610	*No mem.*
GL7d14	8779.760	**1815.540**	*No data*	*Error*
GL7d15	**81012.570**	*No data*	*No data*	*Error*
GL7d23	56805.370	**4202.270**	*No data*	*No mem.*
GL7d24	3511.740	**62.580**	1754.050	143.369
GL7d25	85.690	0.240	0.970	**0.056**
Grobner				
HFE18_96.in.sms	135.810	12.700	11.250	**3.124**
c8_mat11	550.190	**54.540**	62.860	*No mem.*

Matrix	LinBox			Rheinfall
	Black Box	Linear Pvt.	No Pvt.	Sparsity Pvt.
c8_mat11.I	570.280	53.060	**37.770**	*No mem.*
f855_mat9	20.500	0.690	0.540	**0.100**
f855_mat9.I	20.130	0.660	2.040	**0.104**
rkat7_mat5	0.090	0.100	0.170	**0.040**
robot24c1_mat5	**0.020**	**0.020**	**0.020**	*Arith. ovf.*
robot24c1_mat5.J	**0.020**	**0.020**	0.070	*No mem.*
Homology				
D6-6	2113.560	37.420	25.130	**0.060**
ch6-6.b2.2400x450	0.010	0.010	0.010	**<4ms**
ch6-6.b3.5400x2400	3.740	0.140	0.110	**0.008**
ch6-6.b4.4320x5400	8.310	**0.280**	0.410	0.504
ch6-6.b5.720x4320	**<4ms**	0.010	**<4ms**	**<4ms**
ch7-6.b2.4200x630	0.030	0.030	0.020	**0.004**
ch7-6.b3.12600x4200	19.610	0.600	0.440	**0.028**
ch7-6.b4.15120x12600	109.900	2.150	1.730	**0.976**
ch7-6.b5.5040x15120	35.080	0.220	0.190	**0.036**
ch7-7.b2.7350x882	0.070	0.070	0.060	**0.008**
ch7-7.b5.35280x52920	1207.840	**134.700**	380.760	1441.114
ch7-8.b2.11760x1176	3.380	0.160	0.120	**0.012**
ch7-8.b3.58800x11760	376.610	21.160	14.760	**0.172**
ch7-8.b4.141120x58800	4979.530	313.210	201.600	**1.216**
ch7-8.b5.141120x141120	13726.270	**918.070**	1725.720	14986.761
ch7-9.b2.17640x1512	10.070	0.290	0.220	**0.020**
ch7-9.b3.105840x17640	998.390	52.730	36.800	**0.256**
ch7-9.b4.317520x105840	23764.940	1084.750	719.950	**2.228**
ch7-9.b5.423360x317520	112350.890	*No data*	5189.950	**842.729**
ch8-8.b2.18816x1568	12.720	0.310	0.240	**0.020**
ch8-8.b3.117600x18816	1198.760	76.920	44.420	**0.296**
ch8-8.b4.376320x117600	33862.820	1572.430	1004.160	**3.212**
ch8-8.b5.564480x376320	139770.720	*No data*	9345.920	**650.113**
cis.n4c6.b13.6300x25605	135.090	0.390	0.260	**0.012**
cis.n4c6.b14.920x6300	0.010	**<4ms**	0.010	**<4ms**
cis.n4c6.b3.5970x1330	2.220	0.060	0.050	**0.008**
cis.n4c6.b4.20058x5970	68.510	0.910	0.690	**0.024**
lutz30-23.b6.1716x3003	2.250	**0.220**	0.430	0.228
m133.b3.200200x200200	**31480.910**	*No data*	*No data*	*No mem.*
mk10.b2.3150x630	0.020	0.020	0.020	**0.004**
mk10.b3.4725x3150	4.620	0.180	0.110	**0.076**
mk10.b4.945x4725	0.010	**<4ms**	0.010	**<4ms**
mk11.b2.6930x990	0.080	0.080	0.060	**0.008**
mk11.b3.17325x6930	58.150	2.160	1.100	**0.080**
mk11.b4.10395x17325	95.250	**5.620**	32.870	47.603
mk11.b4.9450x17325	73.140	**1.130**	6.920	17.137
mk12.b2.13860x1485	7.220	0.250	0.170	**0.020**
mk12.b3.51975x13860	388.290	20.960	14.960	**0.152**
mk12.b4.62370x51975	2096.490	**134.320**	175.380	2121.453
mk12.b5.10395x62370	9802.160	0.740	0.620	**0.012**
mk13.b5.135135x270270	29762.450	**7071.310**	*No data*	*No mem.*
mk9.b3.945x1260	**0.010**	0.020	0.040	0.016
n2c6.b3.1365x455	**<4ms**	0.010	**<4ms**	**<4ms**
n2c6.b4.3003x1365	1.340	0.030	0.020	**<4ms**
n2c6.b5.4945x3003	5.550	0.140	0.080	**0.008**
n2c6.b6.5715x4945	11.850	0.320	0.130	**0.008**
n2c6.b7.3990x5715	10.240	0.170	0.090	**0.008**

Matrix	LinBox			Rheinfall
	Black Box	Linear Pvt.	No Pvt.	Sparsity Pvt.
n2c6.b8.1470x3990	2.980	0.020	0.020	**<4ms**
n3c6.b10.675x2511	**<4ms**	0.010	**<4ms**	**<4ms**
n3c6.b4.3003x1365	1.340	0.020	0.020	**<4ms**
n3c6.b5.5005x3003	5.360	0.090	0.080	**0.004**
n3c6.b6.6435x5005	13.250	0.160	0.130	**0.004**
n3c6.b7.6435x6435	19.050	0.190	0.150	**0.008**
n3c6.b8.4935x6435	16.840	0.200	0.110	**0.008**
n3c6.b9.2511x4935	6.420	0.090	0.040	**0.008**
n4c5.b4.2852x1350	1.260	0.040	0.020	**<4ms**
n4c5.b5.4340x2852	4.510	0.090	0.060	**0.008**
n4c5.b6.4735x4340	8.420	0.160	0.090	**0.004**
n4c5.b7.3635x4735	7.590	0.100	0.060	**0.004**
n4c5.b8.1895x3635	3.410	0.030	0.020	**0.004**
n4c5.b9.630x1895	0.010	**<4ms**	**<4ms**	**<4ms**
n4c6.b10.132402x186558	23799.010	410.290	228.890	**0.468**
n4c6.b11.69235x132402	7540.700	106.670	65.530	**0.276**
n4c6.b12.25605x69235	1669.380	10.870	5.550	**0.068**
n4c6.b13.6300x25605	153.410	0.370	0.260	**0.012**
n4c6.b14.920x6300	0.010	0.020	**<4ms**	**<4ms**
n4c6.b3.5970x1330	2.220	0.060	0.050	**0.008**
n4c6.b4.20058x5970	66.460	1.380	0.710	**0.028**
n4c6.b5.51813x20058	659.500	22.490	16.050	**0.088**
n4c6.b6.104115x51813	3680.810	163.330	84.550	**0.228**
n4c6.b7.163215x104115	11833.480	363.790	232.310	**0.412**
n4c6.b8.198895x163215	25073.760	727.020	400.850	**0.508**
n4c6.b9.186558x198895	35794.070	961.120	395.730	**0.604**
shar_te2.b1.17160x286	0.040	0.040	0.030	**0.012**
shar_te2.b2.200200x17160	2102.770	136.620	66.460	**1.988**
shar_te2.b3.200200x200200	**26275.470**	*No data*	*No data*	*Error*
Kocay				
Trec10	**<4ms**	**<4ms**	0.010	**<4ms**
Trec11	0.030	0.030	0.110	**0.012**
Trec12	0.420	0.580	1.840	**0.232**
Trec13	116.470	8.640	28.850	**5.820**
Trec14	1249.860	137.750	564.810	**107.679**
Margulies				
cat_ears_2_4	1.260	0.010	0.010	**<4ms**
cat_ears_3_4	33.790	0.380	0.240	**0.012**
cat_ears_4_4	582.930	7.320	4.660	**0.088**
flower_4_4	5.280	0.050	0.060	**0.012**
flower_5_4	43.140	0.440	0.560	**0.048**
flower_7_4	1312.790	21.190	48.740	**3.144**
flower_8_1	**<4ms**	0.010	**<4ms**	**<4ms**
flower_8_4	4879.440	92.810	287.090	**16.077**
kneser_10_4_1	54558.460	2009.800	1552.360	**3.896**
kneser_8_3_1	77.030	1.860	1.510	**0.044**
wheel_601	*No data*	*No data*	**11215.570**	*Error*
Relat				
rel6	**<4ms**	0.010	0.010	0.008
rel7	7.120	**0.690**	1.330	2.128
rel8	1631.420	**268.830**	530.830	3610.130
relat3	**<4ms**	**<4ms**	**<4ms**	*Error*

Matrix	LinBox			Rheinfall
	Black Box	Linear Pvt.	No Pvt.	Sparsity Pvt.
relat4	**<4ms**	**<4ms**	**<4ms**	*Error*
relat5	**<4ms**	**<4ms**	**<4ms**	*Error*
relat6	**0.010**	**0.010**	**0.010**	*Error*
relat7	6.070	**0.640**	1.500	*Error*
relat7b	5.860	**0.640**	1.490	*Error*
relat8	2105.040	**282.230**	701.710	*Error*
Smooshed				
olivermatrix.1	*No data*	0.680	**0.410**	*Error*
olivermatrix.2	88650.570	152.820	83.530	**1.608**
Taha				
mat2_abtaha	0.190	0.340	0.280	**0.028**
mat_abtaha	0.050	0.050	0.040	**0.012**
diffGL6				
D_10	**<4ms**	**<4ms**	0.010	**<4ms**
D_6	**<4ms**	**<4ms**	0.020	0.004
D_7	**0.010**	**0.010**	0.240	0.024
D_8	**<4ms**	0.010	0.250	0.024
D_9	0.010	**<4ms**	0.080	0.004
diffSL6				
D_10	**0.020**	**0.020**	0.360	0.024
D_11	**<4ms**	**<4ms**	0.010	0.004
D_6	**0.020**	**0.020**	0.310	0.024
D_7	**0.070**	**0.070**	2.720	0.260
D_8	0.880	**0.090**	2.890	0.272
D_9	**0.040**	0.050	1.780	0.140

Bibliography

[1] E. Arbarello, M. Cornalba, and P. Griffiths. *Geometry of Algebraic Curves vol. II*, volume 268 of *Grundlehren der mathematischen Wissenschaften*. Springer-Verlag, Heidelberg, 2011. with a contribution by J. Harris.

[2] Enrico Arbarello and Maurizio Cornalba. Divisors in the moduli spaces of curves. In Lizhen Ji, Scott Wolpert, and Shing-Tung Yau, editors, *Geometry of Riemann surfaces and their moduli spaces*, volume 14 of *Surveys in Differential Geometry*, pages 1–22. International Press, Sommerville, MA, 2009. Pre-print available online at: http://arxiv.org/abs/0810.5373v2.

[3] Enrico Arbarello and Maurizio Cornalba. Jenkins-Strebel differentials. *Rend. Lincei Mat. Appl.*, 21:115–157, 2010.

[4] Edward A. Bender and E. Rodney Canfield. The asymptotic number of maps on a surface. *J. Combin. Theory, Ser. A*, 43(2):244–257, 1986.

[5] Edward A. Bender, Zhicheng Gao, and L. Bruce Richmond. The map asymptotics constant t_g. *Electron. J. Combinatorics*, 15, 2008.

[6] J. Bergström and O. Tommasi. The rational cohomology of $\overline{M}_4$. *Mathematische Annalen*, 338(1):207–239, 2007.

[7] G. Bini, G. Gaiffi, and M. Polito. A formula for the Euler characteristic of $\overline{\mathrm{M}}_{2,n}$. *Mathematische Zeitschrift*, 236(3):491–523, 2001.

[8] Gilberto Bini and John Harer. Euler characteristics of moduli spaces of curves. *Journal of the European Mathematical Society*, 13(2):487–512, 2011.

[9] Béla Bollobás and Oliver Riordan. A polynomial of graphs on surfaces. *Math. Ann.*, 323(1):81–96, 2002.

[10] Kenneth S. Brown. *Cohomology of Groups*. Number 87 in Graduate Texts in Mathematics. Springer-Verlag, 1982.

[11] James Conant and Karen Vogtmann. On a theorem of Kontsevich. *Algebr. Geom. Topol.*, 3:1167–1224 (electronic), 2003.

[12] Wikipedia contributors. Python (programming language). Online article at: http://en.wikipedia.org/wiki/Python_(programming_language), July 2009.

[13] I. S. Duff, A. M. Erisman, and J. K. Reid. *Direct Methods for Sparse Matrices*. Clarendon Press, Oxford, 1989.

[14] J.-G. Dumas, T. Gautier, M. Giesbrecht, P. Giorgi, B. Hovinen, E. Kaltofen, B. D. Saunders, W. J. Turner, and G. Villard. LinBox: A Generic Library for Exact Linear Algebra. In A. Cohen, X.-S. Gao, and N. Takayama, editors, *Mathematical Software: ICMS 2002 (Proceedings of the first International Congress of Mathematical Software*, pages 40–50. World Scientific, 2002.

[15] Jean-Guillaume Dumas. The Sparse Integer Matrices Collection. http://ljk.imag.fr/membres/Jean-Guillaume.Dumas/simc.html.

[16] Jean-Guillaume Dumas and Gilles Villard. Computing the rank of large sparse matrices over finite fields. In *CASC'2002 Computer Algebra in Scientific Computing*, pages 22–27. Springer-Verlag, 2002.

[17] GC3Pie website. http://gc3pie.googlecode.com/.

[18] E. Getzler and M. M. Kapranov. Modular operads. *Compositio Math.*, 110(1):65–126, 1998.

[19] E. Getzler and E. Looijenga. The Hodge polynomial of $\overline{M}_{3,1}$. *Arxiv preprint math.AG/9910174*, 1999.

[20] Ezra Getzler. *The moduli space of curves*, volume 129 of *Progr. Math.*, chapter Operads and moduli spaces of genus 0 Riemann surfaces, pages 199–230. Birkäuser, Boston, 1995. Pre-print available online at: http://arxiv.org/abs/alg-geom/9411004.

[21] Ezra Getzler. Topological recursion relations in genus 2. In *Integrable systems and algebraic geometry (Kobe/Kyoto, 1997)*, pages 73–106, River Edge, NJ, 1998. World Scientific Publishing. Pre-print available at: http://arXiv.org/abs/math/9801003.

[22] Ezra Getzler. Resolving mixed Hodge modules on configuration spaces. *Duke Math. J.*, 96(1):175–203, 1999. Pre-print available online at: http://arxiv.org/abs/alg-geom/9611003.

[23] Véronique Godin. The unstable integral homology of the mapping class groups of a surface with boundary. *Math. Ann.*, 337(1):15–60, 2007.

[24] G. Golub and C. Van Loan. *Matrix Computation*. Johns Hopkins University Press, 2nd edition, 1989.

[25] Alexei G. Gorinov. Rational cohomology of the moduli space of pointed genus 1 curves. Pre-print available online at: http://www.liv.ac.uk/~gorinov/.

[26] Laura Grigori, James W. Demmel, and Xiaoye S. Li. Parallel symbolic factorization for sparse LU with static pivoting. *SIAM J. Scientific Computing*, 29(3):1289–1314, 2007.

[27] Alastair Hamilton and Andrey Lazarev. Characteristic classes of A_∞-algebras, 2008.

[28] J. Harer and D. Zagier. The Euler characteristic of the moduli space of curves. *Invent. Math.*, 85(3):457–485, 1986.

[29] John Harer. The second homology group of the mapping class group of an orientable surface. *Invent. Math.*, 72(2):221–239, 1983.

[30] John L. Harer. The virtual cohomological dimension of the mapping class group of an orientable surface. *Invent. Math.*, 84(1):157–176, 1986.

[31] John L. Harer. The cohomology of the moduli space of curves. In *Theory of moduli (Montecatini Terme, 1985)*, pages 138–221. Springer, Berlin, 1988.

[32] Klaus Hulek and Orsola Tommasi. Cohomology of the second Voronoi compactification of $\mathcal{A}_4$, 2011. Pre-print available at: http://arxiv.org/abs/1103.6169.

[33] Kiyoshi Igusa. Combinatorial Miller-Morita-Mumford classes and Witten cycles. *Algebr. Geom. Topol.*, 4:473–520, 2004.

[34] Kiyoshi Igusa. Graph cohomology and kontsevich cycles. *Topology*, 43:1469–1510, 2004.

[35] Intel Corp. Intel Threading Building Blocks for Open Source. http://threadingbuildingblocks.org/.

[36] Intel Corp. Intel Threading Building Blocks v4.0 Reference Manual, 2012. Available at: http://threadingbuildingblocks.org/uploads/81/91/Latest%20Open%20Source%20Documentation/Reference.pdf.

[37] J. A. Jenkins. On the existence of certain extremal metrics. *Annals of Mathematics*, 66:440–453, 1957.

[38] Erich Kaltofen and B. David Saunders. On Wiedemann's method of solving sparse linear systems. In Harold F. Mattson, Teo Mora, and T. R. N. Rao, editors, *Proceedings of the Applied Algebra, Algebraic Algorithms and Error-Correcting Codes 9th International Symposium, AAECC-9 New Orleans, LA, USA*, volume 539 of *Lecture Notes in Computer Science*, Berlin / Heidelberg, October 7–11 1991. Springer-Verlag.

[39] Maxim Kontsevich. Formal (non)commutative symplectic geometry. In *The Gel′fand Mathematical Seminars, 1990–1992*, pages 173–187. Birkhäuser Boston, Boston, MA, 1993.

[40] Maxim Kontsevich. Feynman diagrams and low-dimensional topology. In *First European Congress of Mathematics, Vol. II (Paris, 1992)*, pages 97–121. Birkhäuser, Basel, 1994.

[41] Samuil L. Krushkal′. *Quasiconformal mappings and Riemann surfaces*. V. H. Winston & Sons, Washington, D.C., 1979. Edited by Irvin Kra [Irwin Kra], Translated from the Russian, A Halsted Press Book, Scripta Series in Mathematics, With a foreword by Lipman Bers.

[42] Sergei K. Lando and Alexander K. Zvonkin. *Graphs on surfaces and their applications*, volume 141 of *Encyclopaedia of Mathematical Sciences*. Springer-Verlag, 2008.

[43] LinBox website. http://linalg.org/.

[44] E. Looijenga. Cohomology of M_3 and M_3^1. In *Mapping class groups and moduli spaces of Riemann surfaces (Göttingen, 1991/Seattle, WA, 1991)*, volume 150 of *Contemp. Math*, pages 205–228, 1993.

[45] Eduard Looijenga. Cellular decompositions of compactified moduli spaces of pointed curves. In *The moduli space of curves (Texel Island, 1994)*, pages 369–400. Birkhäuser Boston, Boston, MA, 1995.

[46] MathOverflow question "Betti numbers of moduli spaces of smooth Riemann surfaces". http://mathoverflow.net/questions/38968/betti-numbers-of-moduli-spaces-of-smooth-riemann-surfaces, September 2010.

[47] Brendan D. McKay. Isomorph-Free Exhaustive Generation. *Journal of Algorithms*, 26(2):306–324, 1998.

[48] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard, 1994.

[49] Message Passing Interface Forum. MPI-2: Extensions to the Message-Passing Interface, 1996.

[50] Gabriele Mondello. Combinatorial classes on $\overline{\mathcal{M}}_{g,n}$ are tautological. *Int. Math. Res. Not.*, 44:2329–2390, 2004.

[51] Gabriele Mondello. Riemann surfaces, ribbon graphs and combinatorial classes. In Athanase Papadopoulos, editor, *Handbook of Teichmüller theory*, volume 2, pages 151–216. European Mathematical Society, 2009.

[52] M. Mulase and M. Penkava. Ribbon graphs, quadratic differentials on Riemann surfaces, and algebraic curves defined over $\overline{\mathbb{Q}}$. *Asian J. Math.*, 2(4):875–919, 1998.

[53] David Mumford. Abelian quotients of the Teichmüller modular group. *J. Analyse Math.*, 18:227–244, 1967.

[54] Riccardo Murri. A novel parallel algorithm for Gaussian Elimination of sparse unsymmetric matrices. In Roman Wyrzykowski, editor, *Proceedings of the PPAM2011 Conference*, volume 7203/7204 of *Lecture Notes in Computer Science*. Springer-Verlag, 2012. Pre-print available at: http://arxiv.org/abs/1105.4136.

[55] Riccardo Murri. Fatgraph algorithms and the homology of the kontsevich complex. Pre-print available online at: http://arxiv.org/abs/1202.1820, February 2012.

[56] R. C. Penner and Douglas J. LaFountain. Cell decomposition and odd cycles on compactified Riemann's moduli space. *arXiv*, 1112.3915, 2011. Available online at: http://arxiv.org/abs/1112.3915.

[57] Robert Clark Penner, Michael Knudsen, Carsten Wiuf, and Joergen Ellegaard Andersen. Fatgraph Models of Proteins. *Proteins*, 277:32, 2009.

[58] Swiss Multi-Science Computing Grid website. http://www.smscg.ch/.

[59] Kurt Strebel. *Quadratic differentials*. Springer-Verlag, Berlin, 1984.

[60] O. Tommasi. Rational cohomology of $M_{3,2}$. *Compos. Math*, 143(4):986–1002, 2007.

[61] Orsola Tommasi. *Geometry of Discriminants and Cohomology of Moduli Spaces*. PhD thesis, Radboud University Nijmegen, 2005. Available online at: http://webdoc.ubn.ru.nl/mono/t/tommasi_o/geomofdia.pdf.

[62] Burt Totaro. Configuration spaces of algebraic varieties. *Topology*, 35(4):1057–1067, 1996.

[63] W.J. Turner. *Black box linear algebra with the LINBOX library*. PhD thesis, North Carolina State University, May 2002. Available online at http://repository.lib.ncsu.edu/ir/handle/1840.16/3025.

[64] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33:103–111, August 1990.

[65] Guido van Rossum et al. The Python 2.7 Language Reference. Online document at: http://docs.python.org/2.7/reference/index.html, February 2012.

[66] J. Vetter and C. Chambreau. mpiP: Lightweight, Scalable MPI profiling. http://www.llnl.gov/CASC/mpiP, 2004.

Acknowledgments

It is a pleasure to thank the many people who made this thesis possible.

I am specially indebted, and would like to express my gratitude, to my advisors Enrico Arbarello and Domenico Fiorenza, and to Gilberto Bini, for their constant encouragement and manifold support during the long preparation of this thesis.

Parallel software development and computational experiments require a large amount of processing power, which in this case was kindly donated by the Informatik Dienste of the University of Zurich, by the Organisch-Chemisches Institut (and particularly the "Theoretical Chemistry and Computational Grid Applications" research group led by Kim Baldridge), and the Swiss Multi-Science Computational Grid project (especially the "Grid Computing Competence Center" directed by Sergio Maffioletti); their support is here gratefully acknowledged. I am doubly indebted to Kim and Sergio for letting me work part-time in order to complete this thesis.

Parts of this thesis have already appeared publicly, in arXiv preprints [55] and in the PPAM2011 conference proceedings [54]; I would like to extend my gratitude to those who provided me with feedback. In particular, I would like to thank Professors Julian Hall and Olaf Schenk, for their interest and suggestions regarding the "Rheinfall" algorithm. I am grateful to Orsola Tommasi and Dan Petersen, who pointed out inaccuracies and suggested improvements in the expository chapter on $\mathcal{M}_{g,n}$ homology.

On a more personal note, I am deeply indebted to Marco and Deny for incessantly pushing me towards this goal despite my protests. Thanks to Valentina, for providing the safe harbor among the many changes that occurred in my life along these years. To all of you, thanks above all for your continued friendship. I also cannot overstate my gratitude to my family, for their love, patience, and guidance, and for letting me err in my own way.

Thanks to Agnese for always being there when it was needed. I would not have made it without you.

Writing this thesis took me much more time that was sane or reasonable, and many other people have supported and encouraged me over the years: a full list would probably take another volume, and I've already grown old writing one. Thank you all; this is...

The End.