



ACMAC's PrePrint Repository

Book of Abstracts of the Minisymposium on Publicly Available Geometric/Topological Software

Menelaos I. Karavelas and Monique Teillaud

Original Citation:

Karavelas, Menelaos I. and Teillaud, Monique
(2012)

Book of Abstracts of the Minisymposium on Publicly Available Geometric/Topological Software.

This version is available at: <http://preprints.acmac.uoc.gr/98/>

Available in ACMAC's PrePrint Repository: February 2014

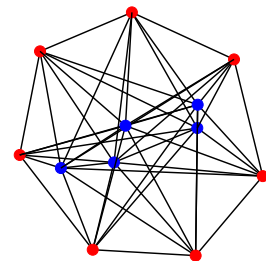
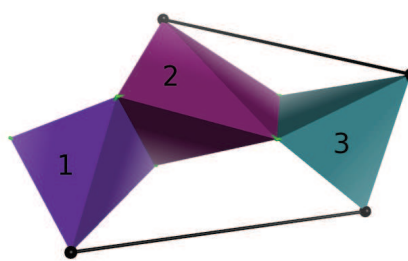
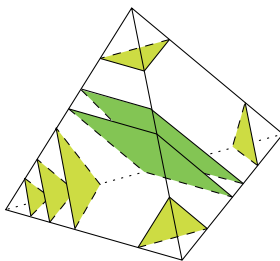
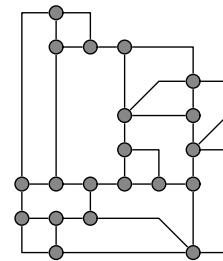
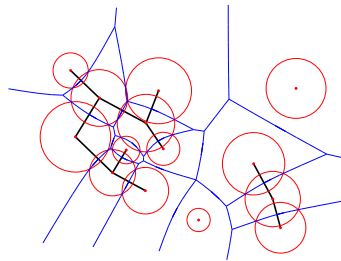
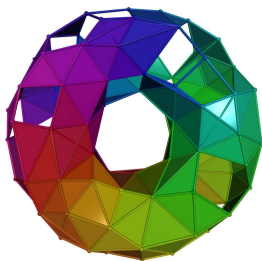
ACMAC's PrePrint Repository aim is to enable open access to the scholarly output of ACMAC.

Book of Abstracts

Minisymposium on Publicly Available Geometric/Topological Software

June 17 & 19, 2012 Chapel Hill, NC, United States

Part of the CG-Week 2012



June 2012

Book of Abstracts
Minisymposium on Publicly Available Geometric/Topological Software
Chapel Hill, NC, United States
June 17 & 19, 2012

Compilation copyright © 2012 by Menelaos Karavelas and Monique Teillaud.
Copyrights of individual papers retained by the authors.

The front page images have been kindly provided by the authors of the corresponding abstracts.
From left to right and top to bottom: Andrew Tausz, Menelaos Karavelas, Carsten Gutwenger,
Benjamin Burton, Ileana Streinu, Michael Joswig.

Preface

In 2012, an expanded set of events was co-located with the 28th Annual Symposium on Computational Geometry (SoCG), hosted at UNC Chapel Hill, June 17-20, 2012. The Minisymposium on Publicly Available Geometric/Topological Software (GTS) was one of them, and was scheduled for the afternoons of June 17th and June 19th.

The aim of the minisymposium was to provide a platform for the presentation of publicly available and/or open source geometric and topological software. It was intended to provide a means for the dissemination and familiarization of existing or forthcoming publicly available software. It consisted of a series of tutorial talks about open source and/or publicly available geometric/topological software, which the attendees were to follow on their own laptops, followed by an open demo/software presentation session.

We had four invited tutorial on software packages for persistent homology, 3-manifold topology, polytope geometry, and Voronoi diagrams. As a result of the open call for contributions, there were six submissions, among which three were accepted as tutorials for software packages on graph drawing, persistent homology and rigidity analysis, two were accepted for the open demo session, and one was rejected.

As organizers, we would like to thank the invited speakers Benjamin Burton, Michael Joswig and Dmitriy Morozov for accepting our invitation, as well as the contributed presenters for submitting a proposal for their software packages. Finally, we would like to thank the SoCG local organizer Jack Snoeyink, and the CG-Week coordinator Joe Mitchell for their help and support in organizing GTS.

Menelaos Karavelas
Monique Teillaud

Organizers

Menelaos Karavelas	University of Crete & FO.R.T.H.
Monique Teillaud	INRIA Sophia Antipolis – Méditerranée

Table of Contents

Sunday, June 17, 2012

14:20 – 15:50 Session 1: Persistent Homology

Invited talk: Dionysus A brief overview 1
Dmitriy Morozov

javaPlex: A research platform for persistent homology 7
Andrew Tausz, Henry Adams, Mikael Vejdemo-Johansson

16:10 – 17:40 Session 2: Manifold Topology & Graph Drawing

Invited talk: Using Regina to experiment and compute with 3-manifold
triangulations and normal surfaces 13
Benjamin Burton

The Open Graph Drawing Framework (OGDF) 19
Carsten Gutwenger

Tuesday, June 19, 2012

15:20 – 16:50 Session 3: Polytope Geometry & Rigidity Analysis

Invited talk: An Introduction to polymake 2.12 25
Michael Joswig

KINARI-Lib: a C++ library for mechanical modeling and pebble game rigidity analysis 29
Naomi Fox, Filip Jagodzinski, Ileana Streinu

17:20 – 18:05 Session 4: Triangulations and Voronoi Diagrams

Invited talk: Solving problems with CGAL: an example using
the 2D Apollonius graph package 33
Menelaos I. Karavelas

18:15 – 18:55 Demo Session

M4G: Manifolds for GPUs Library 39
André Maximo, Luiz Velho

GAP persistence – a computational topology package for GAP 43
Mikael Vejdemo-Johansson

Dionysus

A brief overview

Dmitriy Morozov*

1 Introduction

Over the last decade, *persistent homology* [2] has matured from an elegant idea into a rich theory with century-old roots. Two pillars — algebra and fast algorithms — support its foundation. The former has guided its theoretical development; the latter has nurtured a range of applications. Software is the essential bridge between the two: assembling the growing collection of techniques in a single code-base makes the theory accessible to a wider audience.

Although Dionysus' original goal was to facilitate experiments with persistence algorithms, we won't discuss it here. Instead we focus on an accidental benefit. While the core library is written in C++, most of its functionality has been exposed to Python. This high-level interface is both rich and simple, allowing for many sophisticated techniques to be implemented in a few lines of code. Because it follows the C++ API very closely, translating a Python prototype into C++ becomes a straightforward exercise. The library is available at mrzv.org/software/dionysus.

Throughout the overview we assume familiarity with persistence — [1] is an excellent introduction — and give only cursory definitions.

2 Functions and filtrations

The main input to any persistence algorithm is a filtration of a simplicial complex, i.e., a nested sequence of complexes:

$$\emptyset = K_0 \subseteq K_1 \subseteq \dots \subseteq K_{n-1} \subseteq K_n = K. \quad (1)$$

We can assume without loss of generality that subsequent complexes differ by a single simplex. So we can represent a filtration compactly by an ordered list of simplices $[\sigma_1, \sigma_2, \dots]$, where complex $K_i = \cup_{j=1}^i \sigma_j$. It's rarely practical to construct such a filtration by hand. So, unless it is generated by an external tool, we rely on a few common constructions. Two of them cover many standard scenarios: distance functions and piecewise-linear functions.

2.1 Distance functions

In computational geometry, a distance function is a fruitful representation of the geometry of a compact set. Denoted by $d_P : \mathbb{R}^d \rightarrow \mathbb{R}$ for a compact set $P \subseteq \mathbb{R}^d$, it measures everywhere the distance to the closest point in P , $d_P(x) = \inf_{y \in P} \|x - y\|$.

Alpha shapes. The sublevel sets of a distance function are unions of balls, $d_P^{-1}(-\infty, r] = \cup_{p \in P} B_r(p)$. When the set P is finite, we can clip these balls by the Voronoi regions of their centers to get a collection $\{B_r(p) \cap \text{Vor}(p)\}_p$. Because the sets are convex, the nerve of this collection, called an *alpha shape*, captures the homotopy type of the union:

$$\text{AS}_r(P) = \text{Nrv}_{p \in P} \{B_r(p) \cap \text{Vor}(p)\} \simeq \cup_{p \in P} B_r(p).$$

As we increase the threshold r , the alpha shapes filter the Delaunay triangulation of the point set P . In particular, we can assign to each simplex the smallest value of the distance function on its dual Voronoi cell. In this context, $\text{AS}_r(P)$ is the union of the simplices for which this value does not exceed r . In Python, we can compute the Delaunay triangulation together with the necessary values with the `fill_alpha_complex()` function. Each

*Lawrence Berkeley National Laboratory, dmitriy@mrzv.org. The author would like to acknowledge the support at LBNL of the DOE Office of Science, Advanced Scientific Computing Research, under award number KJ0402-KRD047, under contract number DE-AC02-05CH11231.

generated simplex stores a pair (value,critical) in its `data` field. Here `value` determines when the simplex enters the alpha shape, and `critical` is a flag indicating whether the simplex intersects its dual Voronoi cell. Consequently, we can use Python list comprehensions to extract any given alpha shape:

```
simplices = Filtration()
fill_alpha_complex(points, simplices)
alpha_shape = [s for s in simplices if s.data[0] <= alpha]
```

Or get a complete alpha-shape filtration by sorting the simplices with respect to the `data` field and `dimension`:

```
simplices.sort(data_dim_cmp)
```

Class `Filtration` is an auxiliary construction that both records the order of the simplices, and allows fast access to any given simplex.

Vietoris–Rips complexes. Often it is more convenient to work with pairwise distances between points, rather than their explicit embedding in a Euclidean space. In this case, an alternative construction approximates the information captured by the distance function. *Vietoris–Rips complex for parameter r* contains all those simplices whose edge lengths do not exceed r , $VR_r(P) = \{\sigma \subseteq P \mid \|u - v\| \leq r \ \forall u, v \in \sigma\}$.

In Python, we express distances between points through an auxiliary class that knows their number and how to measure a distance between a pair, see the example below. (`PairwiseDistances` provides a shortcut for points in a Euclidean space.) Given an instance `distances` of such a class, we can instantiate a class `Rips` whose method `generate` fills a simplicial complex up to prescribed threshold r and skeleton dimension k . Furthermore, its method `cmp` is suitable for sorting the simplices by increasing edge lengths, while `eval` determines for any simplex the length of its longest edge.

The following example constructs a `DictDistances` class that looks up edge lengths in a dictionary, returning infinity for missing information. It's then used as an input to the `Rips` class to generate a filtration of a 2-skeleton of a Vietoris–Rips complex for parameter $r = 3$.

```
class DictDistances:
    def __init__(self, dictionary, count = None):
        self.d = dictionary
        self.count = count or len(dictionary)

    def __call__(self,i,j):
        if i in self.d and j in self.d[i]:
            return self.d[i][j]
        elif j in self.d and i in self.d[j]:
            return self.d[j][i]
        else:
            return float('inf')

    def __len__(self):
        return len(self.d)

octahedron = {0: {1: 1, 3:1}, 2: {1:1, 3:1}, 4: {0:2, 1:2, 2:2, 3:2}, 5: {0:3, 1:3, 2:3, 3:3}}
distances = DictDistances(octahedron, 6)
rips = Rips(distances)
simplices = Filtration()
rips.generate(2, 3, simplices.append)      # 2-skeleton up to distance 3
simplices.sort(rips.cmp)

for s in simplices:
    print s, rips.eval(s)
```

2.2 Piecewise-linear functions

Given a simplicial complex K and a function $\hat{f} : \text{Vert } K \rightarrow \mathbb{R}$ defined on the vertices of K , we linearly interpolate it on the interiors of the simplices. The result is a piecewise-linear function $f : K \rightarrow \mathbb{R}$. The sublevel sets of

f are not subcomplexes of K — a levelset may cut a simplex in half — so we need a scheme to construct a filtration with the same persistence as f . To solve this problem, we define a *lower-star filtration* of K by setting $K_a = \{\sigma \in K \mid \max_{v \in \sigma} \hat{f}(v) \leq a\}$. In words, the subcomplex K_a contains all the simplices on which the function f does not exceed value a . It's clear from the definition that K_a changes only as a passes the value $\hat{f}(v)$ of one of the vertices. It's not immediately obvious, but true that K_a is homotopy equivalent to $f^{-1}(-\infty, a]$.

Assuming `list values` contains the function \hat{f} , where `values[i] = $\hat{f}(v_i)$` , we construct the lower-star filtration of $f : K \rightarrow \mathbb{R}$ by sorting the simplices with respect to their highest vertex:

```
def max_vertex_cmp(values):
    def max_vertex(s):
        return max(values[v] for v in s.vertices)

    def compare(si,sj):
        return cmp(max_vertex(si), max_vertex(sj)) or cmp(si.dimension(), sj.dimension())

    return compare

filtration = Filtration(complex, max_vertex_cmp(values))
```

`max_vertex_cmp()` returns a comparison function that remembers the list of vertex `values`. Sorting the filtration, we get the lower-star filtration ordering of all the simplices.

3 Persistence algorithms and diagrams

Given a filtration (1), we can compute its homology as a sequence of vector spaces and linear maps connecting them (we assume homology is computed with coefficients in a field):

$$0 \rightarrow H(K_1) \rightarrow \dots \rightarrow H(K_{n-1}) \rightarrow H(K_n).$$

Persistence keeps track of cycles as they appear and disappear in this filtration and computes a collection of intervals that represent the cycles' lifetimes. We can think of the input to a persistence algorithm as the boundary matrix D , where the columns and rows are ordered with respect to the filtration. In turn, we can interpret standard persistence algorithms as computing a matrix decomposition $R = DV$, where matrix R is reduced, meaning the lowest non-zero entries in its columns fall in unique rows, and matrix V is full-rank and upper-triangular.

The columns of these auxiliary matrices have an immediate interpretation. If the column $R[i] = 0$, then the column $V[i]$ is, by definition, a cycle; it's born with the addition of σ_i in the filtration. If $R[i] \neq 0$, then it's a cycle that dies with the addition of simplex σ_i . Indeed, it's the boundary of the chain $V[i]$, which contains σ_i . The row j of the lowest non-zero entry in column $R[i]$ tells us when the cycle was born — namely, with the addition of simplex σ_j . Notice that matrix R contains all the pairing information.

3.1 Algorithms

In Dionysus, `StaticPersistence` expresses the original algorithm [2], which computes matrix R . If we are interested in both matrices, R and V , we can use the class `DynamicPersistenceChains`. Both have a method `pair_simplices`, which performs the actual computation¹.

```
p = StaticPersistence(filtration)
p.pair_simplices()

or

p = DynamicPersistenceChains(filtration)
p.pair_simplices()
```

Once the matrices are processed, we can examine their result by iterating over the instances of these classes. An auxiliary map `smap` converts the iterators into the filtration simplices.

¹`StaticPersistence.pair_simplices()`, by default, uses an optimization [2] that stores only positive simplices in matrix R . This behavior can be turned off by passing `True` to the method.

```

smap = p.make_simplex_map(filtration)
for i in p:
    if i.sign():
        sigma = smap[i]
        if i.unpaired():
            print sigma.dimension(), sigma.data, "inf"
        else:
            tau = smap[i.pair()]
            print sigma.dimension(), sigma.data, tau.data
    
```

In the `for`-loop, the `sign()` of a simplex determines whether it created a homology class or destroyed it. In the former case, we recover the simplex `sigma` responsible for the birth. We check whether it is `unpaired()`, i.e., whether the homology class it created ever died. If it didn't, we output the `dimension()` and `data` associated with `sigma` as well as "inf" to indicate the infinite lifetime. If `sigma` is paired, we recover its `pair()` `tau`, and output `sigma`'s `dimension()` and `data` as the birth time, plus `tau`'s `data` as the death time.

The above snippet outputs the persistence diagram — in fact, all the non-empty persistence diagrams. But sometimes we want more. To recover explicit cycles that were born and later died in the filtration, i.e., the columns of matrix R , we use the `cycle` attribute of the persistence elements. The following snippet outputs the matrix R , one column per line.

```

for i in p:
    for ii in i.cycle:
        print smap[ii],
    print
    
```

If `p` is an instance of `DynamicPersistenceChains`, its elements have an attribute `chain`, which gives access to columns of matrix V . Thus we can, for example, recover the cycles that never die:

```

for i in p:
    if i.sign() and i.unpaired():
        for ii in i.chain:
            print smap[ii],
        print
    
```

Remark. Dionysus also provides `ZigzagPersistence` and `CohomologyPersistence` classes with slightly different semantics. We omit them here for lack of space.

3.2 Diagram comparison

It's convenient to record a collection of all persistence pairs (b_i, d_i) in a *persistence diagram*, denoted by $\text{Dgm}_p(f)$. Function `init_diagrams()` returns all such (non-zero) diagrams, each expressed in a class `PersistenceDiagram`.

```

diagrams = init_diagrams(p, filtration)
    
```

A fundamental property of persistence is its stability [3], which says that if two functions are close, then so are their persistence diagrams:

$$W_\infty(\text{Dgm}_p(f), \text{Dgm}_p(g)) \leq \|f - g\|_\infty.$$

Here W_∞ denotes the bottleneck distance between the two diagrams. To define it, we compute a bijection $\gamma : \text{Dgm}_p(f) \rightarrow \text{Dgm}_p(g)$ that minimizes the longest distance between a point and its image:

$$W_\infty(\text{Dgm}_p(f), \text{Dgm}_p(g)) = \inf_{\gamma} \sup_x \|x - \gamma(x)\|_\infty.$$

Rephrased algorithmically, we find the smallest value ε such that the bipartite graph on the two diagrams with an edge for every pair of points closer than ε contains a perfect matching. In Dionysus, given two diagrams `dgm1` and `dgm2`, we can find their `bottleneck_distance(dgm1, dgm2)`.

Stronger stability results hold for more restrictive classes of functions. Specifically, for Lipschitz functions (with certain conditions on their domains), the persistence diagrams are stable with respect to the Wasserstein distance [4]:

$$W_q(\text{Dgm}_p(f), \text{Dgm}_p(g)) \leq C \cdot \|f - g\|_\infty^k,$$

where constants C and k depend on the domain of the functions. Here the Wasserstein distance finds the matching γ that minimizes the sum of powers of its edge lengths:

$$W_q(\text{Dgm}_p(f), \text{Dgm}_p(g)) = \left(\inf_{\gamma} \sum_x \|x - \gamma(x)\|_{\infty}^q \right)^{1/q}.$$

The seemingly small change makes a major difference in practice: W_q is much more sensitive because it accounts for the full diagrams rather than just the longest edge in a matching. In Dionysus, we can find it by calling `wasserstein_distance(dgm1, dgm2, q)`.

4 Code

Derived metric. To illustrate some of the above features, consider the following problem. Given a collection of 3-dimensional point clouds, stored in a list `point_clouds`, we want to find the persistence diagrams of the distance functions they induce on \mathbb{R}^3 . Treating these diagrams as points, we measure the Wasserstein distances between them. These measurements define a (finite) metric space, and we compute the persistence of its Vietoris–Rips filtration. To put it briefly: what is the homology of the collection of point clouds?

First we compute the persistence diagrams for the alpha shape filtration for every point set in the collection:

```
def alpha_persistence_diagrams(points):
    f = Filtration()
    fill_alpha_complex(points, f)
    f.sort(data_dim_comparison)
    p = StaticPersistence(f)
    p.pair_simplices()
    diagrams = init_diagrams(p, f, lambda s: s.data[0])
    return diagrams

diagrams = [alpha_persistence_diagrams(cloud) for cloud in point_clouds]
```

Armed with the diagrams, we create a `WassersteinDistances` class suitable for the construction of a Vietoris–Rips filtration. We pick arbitrary thresholds `k` and `r`; their real choice is very application-dependent.

```
class WassersteinDistances:
    def __init__(self, diagrams, q, dim = 1):
        self.diagrams = diagrams
        self.dimension = dim
        self.q = q

    def __len__(self):
        return len(self.diagrams)

    def __call__(self, i, j):
        dgm_i = self.diagrams[i][self.dimension]
        dgm_j = self.diagrams[j][self.dimension]
        return wasserstein_distance(dgm_i, dgm_j, self.q)

distances = WassersteinDistances(diagrams, q = 3)
rips = Rips(distances)
simplices = Filtration()
rips.generate(k, r, simplices.append)      # k-skeleton up to distance r
simplices.sort(rips.cmp)
```

Having generated the Vietoris–Rips filtration `simplices`, we compute its persistence diagrams:

```
p = StaticPersistence(simplices)
p.pair_simplices()
rips_diagrams = init_diagrams(p, simplices, rips.eval)
```

Besides pure curiosity, the knowledge of homology helps us choose candidate target spaces for dimensionality reduction. For example, if we find a 1-dimensional homology class, we can translate it into a map to a circle [5].

Noisy domain. Stability of persistence diagrams helps us cope with the noise in the function values when the domain is fixed. But sometimes even the domain can be noisy. For example, instead of a function $f : X \rightarrow \mathbb{R}$ on a perfect compact set X , we may be given its noisy sampling P , where each point has a value $\hat{f}(p)$. With mild assumptions on the function — most importantly, it needs to be Lipschitz — and the quality of the sampling, we can recover the persistence diagram of the implicit original $f : X \rightarrow \mathbb{R}$ using *image persistence*. Two papers propose the same procedure [6, 7]: the first using alpha-shapes, the second using Vietoris–Rips complexes.

Denoting the Vietoris–Rips complex $\text{VR}_y(P)$ with P^y , the key idea is that a map on homology from P^α to P^β , for suitably chosen α and β , filters out noisy homology classes while preserving the real cycles of X . (It’s exactly the idea behind homology inference [3].) If now we filter these sublevel sets with respect to the function value f and look at the images of the former filtration in the latter, we get an image persistence diagram that provably approximates the persistence diagram of the original function. Denoting by P_x^y the Vietoris–Rips complex for parameter y constructed on the points p with $\hat{f}(p) \leq x$, i.e., $P_x^y = \text{VR}_y(\hat{f}^{-1}(-\infty, x])$, we get a pair of persistence modules:

$$\begin{array}{ccccccc} H(P_{a_1}^\beta) & \rightarrow & H(P_{a_2}^\beta) & \rightarrow & \dots & \rightarrow & H(P_{a_{n-1}}^\beta) & \rightarrow & H(P_{a_n}^\beta) \\ \uparrow & & \uparrow & & & & \uparrow & & \uparrow \\ H(P_{a_1}^\alpha) & \rightarrow & H(P_{a_2}^\alpha) & \rightarrow & \dots & \rightarrow & H(P_{a_{n-1}}^\alpha) & \rightarrow & H(P_{a_n}^\alpha) \end{array}$$

The persistence diagram of the image of the lower module in the upper approximates the persistence diagram of f . To express this pair of filtrations in Dionysus, we must filter P^β with respect to the sampled function \hat{f} , and indicate when a simplex falls into P^α .

```
# Assume points and values are given as well as hdim, alpha, and beta
distances = PairwiseDistances(points)
rips      = Rips(distances)
simplices = Filtration()
rips.generate(hdim + 1, beta, simplices.append)
simplices.sort(max_vertex_cmp(values))           # Just like a PL-function

img_persistence = ImagePersistence(simplices, lambda s: rips.eval(s) <= alpha)
img_persistence.pairsimplices()
```

The `lambda`-function passed to `ImagePersistence` indicates for every simplex whether it belongs to the sub-complex P^α .

References

- [1] HERBERT EDELSBRUNNER AND JOHN HARER. *Computational Topology. An Introduction*. American Mathematical Society, Providence, Rhode Island, 2010.
- [2] HERBERT EDELSBRUNNER, DAVID LETSCHER, AND AFRA ZOMORODIAN. Topological Persistence and Simplification. *Discrete and Computational Geometry*, **28**:511–533, 2002.
- [3] DAVID COHEN-STEINER, HERBERT EDELSBRUNNER, AND JOHN HARER. Stability of Persistence Diagrams. *Discrete and Computational Geometry*, **37**:103–120, 2007.
- [4] DAVID COHEN-STEINER, HERBERT EDELSBRUNNER, JOHN HARER, AND YURIY MILEYKO. Lipschitz Functions Have L_p -Stable Persistence. *Foundations of Computational Mathematics*, **10**:127–139, 2010.
- [5] VIN DE SILVA, DMITRIY MOROZOV, MIKAEL VEJDEMO-JOHANSSON. Persistent Cohomology and Circular Coordinates. *Discrete and Computational Geometry*, **45**:737–759, 2011.
- [6] DAVID COHEN-STEINER, HERBERT EDELSBRUNNER, JOHN HARER, AND DMITRIY MOROZOV. Persistent Homology for Kernels, Images, and Cokernels. Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, pages 1011–1020, 2009.
- [7] FRÉDÉRIC CHAZAL, LEONIDAS J. GUIBAS, STEVE Y. OUDOT, PRIMOZ SKRABA. Scalar Field Analysis over Point Cloud Data. *Discrete and Computational Geometry*, **46**:743–775, 2011.

javaPlex: a research platform for persistent homology

Andrew Tausz*

Mikael Vejdemo-Johansson†

Henry Adams‡

Abstract

The `javaPlex` software package continues the Stanford tradition of software for persistent homology and cohomology computation. `javaPlex` in particular is built with explicit aims for ease of use as a tool for research into computational topology, and is available under an open source license with extensive source code documentation.

The main design aim in the construction of `javaPlex` has been ease of extension – lowering the threshold to implement new algorithms or insights from ongoing research into computational topology in a framework that provides most if not all current persistence algorithms in a single interface.

1 Motivation and Design Goals

The main reason for the existence of `javaPlex` is to provide researchers in the area of topological data analysis a unified software library to support their investigations. With this in mind, the design goals for it are as follows:

- **Support for new directions for research:** The main goal of the `javaPlex` package is to provide an extensible base to support new avenues for research in computational homology and data analysis. While its predecessor `jPlex` was very well suited towards computing simplicial homology, its design made extension difficult.
- **Interoperability:** `javaPlex` can be run either as a Java application, or it can be called from any Java compatible language. Supported languages include Matlab, Jython and Processing.
- **Adherence to generally accepted software engineering practices:** As a means to realizing the first goal, the `javaPlex` software package was designed and implemented with software engineering best-practices. Emphasis was placed on maintainability, modularity and reusability of the different parts of the code.

We refer the reader to [Car09] for a very readable introduction to the field of topological data analysis as well as the computational tasks involved.

2 Availability and Licensing

`javaPlex` is an open source package, available through Google Code at <http://code.google.com/p/javaplex>. The `javaPlex` package is issued under the New BSD License which is describe at <http://www.opensource.org/licenses/bsd-license.php>.

3 Related Platforms

Two platforms similar in capability and spirit to `javaPlex` are Dionysus and GAP persistence. Dionysus contains state-of-the-art implementations of the standard algorithms for persistent topology and topological data analysis. While Dionysus may have an edge in terms of computational speed (especially since it is written in C++), `javaPlex` is written so that it is maximally accessible to newcomers to the field of computational topology. GAP persistence is a new package developed by one of the authors which presents similar algorithms in the context of the group-theoretic computer algebra system GAP.

*ICME, Stanford University, atausz@stanford.edu

†School of Computer Science, University of St Andrews, Scotland , mvvj@st-andrews.ac.uk

‡Department of Mathematics, Stanford University, henrya@math.stanford.edu

3.1 Overall Structure

In a nutshell, the two main capabilities of `javaPlex` are:

- Computation of the persistent homology of filtered chain complexes of finite vector spaces
- Automated construction of filtered complexes from geometric data

We expect most users of `javaPlex` to go through the “standard” persistent homology pipeline: compute persistent simplicial homology of complexes constructed from point cloud data using Vietoris-Rips or witness complexes. The internal structure, however, anticipates and includes further extensions.

4 Algebraic Background on Persistence Modules

To compute the homological properties of a point cloud, [ELZ02] introduced persistent homology, refined by [ZC05]. Using one of a whole family of methods, the point cloud induces a filtered simplicial complex, where the filtration encodes distance data as increasing “closeness” data for the data points in the point cloud.

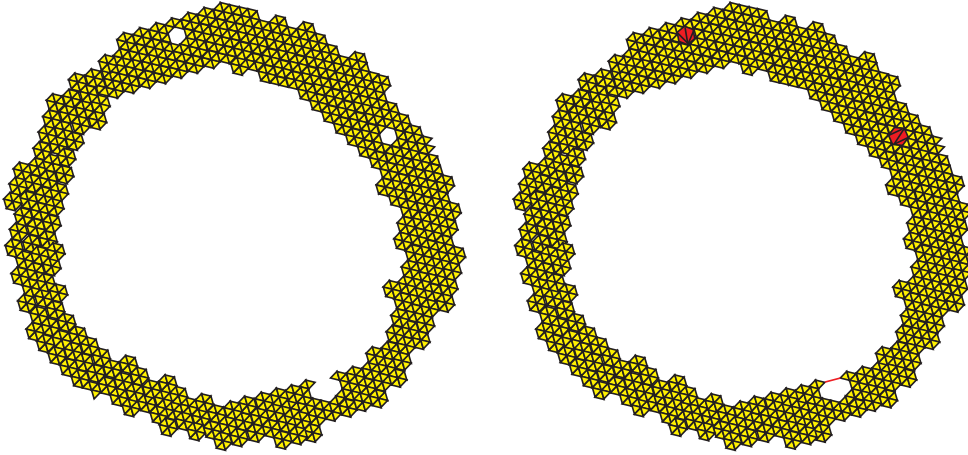


Figure 1: Motivating the definition of persistence: the homology classes that survive from one stage to the next are more likely to be large and significant; all the small, potentially noise-induced classes exist in only one of these two stages.

Thus, from a point cloud, we acquire a filtered simplicial complex. The homology of this with field coefficients is, since homology is a functor, a linear diagram of vector spaces and linear maps.

Consider the situation in Figure 1. To the left, we have a hypothetical intermediate stage of a filtered simplicial complex. The betti number at this stage, an indicator of the size of the homology of a geometric object, would indicate a structure with three holes. However, as the filtration continues, two of the three detected holes are filled in, yielding the picture to the right. However, the gap to the lower right ends up bridged, creating two holes, both detected by the homology computation.

By a cursory inspection of the point cloud, we would be inclined to say that the “true” number of relevant holes, or cycles, in the data is just one – and this one significant hole is characterized by the homology class corresponding to the hole being in the image of the inclusion map: the two superfluous cycles to the left vanish along the map induced by the inclusion of simplicial complexes, and the superfluous cycle to the left was not present before this induced map.

This example motivates our definition of a *persistent homology class* with life time starting at the filtration stage p and *persisting* through to filtration stage q as an equivalence class of cycles in the simplicial complex Σ_p modulo boundaries from the simplicial complex Σ_q .

The dimensions of such persistent homology groups are captured as *betti barcodes*, in which a line is drawn parallel to the x -axis, from p to q for any basis element in this $H_*^{p,q}(\Sigma_*)$. Similarly, a *persistence diagram* captures the same information, with each basis element corresponding to the point (p, q) in the plane. We view as significant features those that are described by long intervals, or equivalently those with a long distance from the $x = y$ diagonal in the persistence diagram.

As demonstrated in [ZC05], the sequential diagrams of linear vector spaces and linear maps

$$V_0 \xrightarrow{i_0} V_1 \xrightarrow{i_1} \dots$$

corresponding to the homology of a filtered simplicial complex correspond through an equivalence of categories to graded modules over $\mathbb{F}[x]$, the polynomial ring in one variable. The equivalence is by sending the diagram above to the vector space $\bigoplus_k V_k$, and introducing an action of x on this vector space by $x \cdot v = i_k(v)$ for $v \in V_k$.

The equivalence carries a long way: the chain complex of a filtered simplicial complex is a filtered chain complex, which fits into this model too, by making each stage correspond to the corresponding filtration stage. Thus, the homology of a filtered simplicial complex becomes a process entirely internal to the category of graded $\mathbb{F}[x]$ -modules.

It should be noted that these modules are graded in two useful ways. One grading comes from the filtration, setting $|v| = k$ for a basis element v if $v \in V_k/V_{k-1}$. Another comes from the dimension: $|v| = k$ if v is a simplicial k -chain.

5 Filtered Complex Generation

As mentioned in the abstract, one of the primary function of `javaPlex` is the construction of filtered chain complexes of vector spaces associated to actual point cloud datasets. The motivation for such constructions is that they provide a persistent model of the dataset in question across all scales. `javaPlex` currently supports the construction of two main types of filtered simplicial complexes: the Vietoris-Rips and lazy-witness constructions. To begin, suppose that we have a finite metric space (\mathcal{X}, d) . In practice, it is possible that \mathcal{X} is a set of points in Euclidean space, although this is not necessary.

5.1 The Vietoris-Rips Construction

We define the filtered complex $\text{VR}(\mathcal{X}, r)$ as follows. Suppose that the points of \mathcal{X} are $\{x_1, \dots, x_N\}$, where $N = |\mathcal{X}|$. The Vietoris-Rips complex is constructed as follows:

- **Add points:** For all points $x \in \mathcal{X}$, $x \in \text{VR}_0(\mathcal{X}, 0)$
- **Add 1-skeleton:** The 1-simplex $[x_i, x_j]$ is in $\text{VR}_1(\mathcal{X}, r)$ iff $d(x_i, x_j) \leq r$
- **Expansion:** We define $\text{VR}(\mathcal{X}, r)$ to be the maximal simplicial complex containing $\text{VR}_1(\mathcal{X}, r)$. That is, a simplex $[x_0, \dots, x_k]$ is in $\text{VR}(\mathcal{X}, r)$ if and only if all of its edges are in $\text{VR}_1(\mathcal{X}, r)$.

An extensive discussion on algorithms for computing the Vietoris-Rips complex can be found in [Zom10]. The `javaPlex` implementation is based on the results of this paper.

5.2 The Lazy-Witness Construction

The fundamental idea behind the lazy-witness construction is that a relatively small subset of a point cloud can accurately describe the shape of the dataset. This construction has the advantage of being more resistant to noise than the Vietoris-Rips construction. An extensive discussion about it can be found in [dSC04].

The lazy-witness construction starts with a selection of landmark points, $\mathcal{L} \subset \mathcal{X}$ with $|\mathcal{L}| = L$. One possibility is to simply choose a random subset of \mathcal{X} . Another possibility is to perform a sequential max-min selection: An initial point l_0 is selected, and then we inductively select the point l_k which maximizes the minimum distance to all previously generated points. This max-min construction tends to produce more evenly spaced points than the random selection. Again we refer the reader to [dSC04] for a more detailed discussion, as well as empirical results supporting these claims.

This construction is parameterized by a value ν , which most commonly takes the values 0, 1, or 2. We also define the distance matrix D to contain the pairwise distances between the points in \mathcal{X} .

- **Define m_i :** If $\nu = 0$, let $m_i = 0$, otherwise, define m_i to be the ν -th smallest entry in the i -th column of D
- **Add points:** For all points $l \in \mathcal{L}$, $l \in \text{LW}_0(\mathcal{X}, 0, \nu)$
- **Add 1-skeleton:** The 1-simplex $[l_i, l_j]$ is in $\text{LW}_1(\mathcal{X}, r, \nu)$ iff there exists an $x \in \mathcal{X}$ such that $\max(d(l_i, x), d(l_j, x)) \leq r + m_i$.
- **Expansion:** We define $\text{LW}(\mathcal{X}, r, \nu)$ to be the maximal simplicial complex containing $\text{LW}_1(\mathcal{X}, r, \nu)$.

6 Homology Computation

The other primary function of the `javaPlex` library is the set of algorithms that actually compute the homology and cohomology of a filtered chain complex. Key references to background material regarding these algorithms can be found in [ZC05, dSMVJ10]. Although we do not describe them in detail here, we note that the algorithms for computing persistent absolute/relative (co)homology can be formulated as matrix decomposition problems. The fundamental reason for this is the equivalence of category of persistent vector spaces of finite type, and the category finitely generated graded modules over $\mathbb{F}[t]$. This correspondence is described in [ZC05].

The homology algorithms are built in a way that is optimized for chain complexes implemented as *streams*. By this we mean that a filtered chain complex is represented by a sequence of basis elements that are produced in increasing order of their filtration indices. Enforcing the constraint that all complexes must be implemented this way allows `javaPlex` to perform the matrix decomposition operations in an efficient online fashion.

7 Applications

Although in principle `javaPlex` can compute the persistent homology of arbitrary chain complexes of vector spaces, almost always these complexes arise from some sort of topological construction. Below we outline these different situations.

7.1 Simplicial Homology

This is the “standard” situation, which was also handled by previous versions in the Plex software family. Here, we have a filtered sequence of simplicial complexes $X_1 \subset X_2 \subset \dots \subset X_n$, from which we define the vector space of chains, $C(X_i)$ consisting of formal sums of elements of X_i with coefficients in the field \mathbb{F} .

In this case the boundary operator $\partial : C(X_i) \rightarrow C(X_{i-1})$ is the actual geometric boundary defined by

$$\partial([v_0, \dots, v_n]) = \sum_i (-1)^i [v_0, \dots, \hat{v}_i, \dots, v_n]$$

7.2 Cellular Homology

In this case, $X = X_*$ is a filtered cell complex. This complex is formed by inductively adding n -cells to the $n - 1$ -skeleton, by the gluing maps

$$\varphi_\alpha^n : S^{n-1} \rightarrow X_{d-1}$$

which map the boundaries of the n -cells e_α^n to the $n - 1$ skeleton. Note that in the above, we use n to denote the grading by dimension, and d to denote the grading by filtration index.

The boundary operator then becomes

$$\partial(e_\alpha^n) = \sum_\beta \deg(\varphi_{\alpha\beta}^n) e_\beta^{n-1}$$

where \deg refers to the topological degree of a map.

Note that while `javaPlex` is fully capable of computing the persistent homology of arbitrary cell complexes, the specification of such complexes are more tedious than simplicial complexes. Nevertheless they offer the user a parsimonious way of defining a wide class of topological spaces.

7.3 Operations on Chain Complexes

The abstraction away from geometric primitives allows `javaPlex` to handle more general algebraic constructions using complexes. These include computations such as tensor products, hom complexes and mapping cylinders.

8 Examples

8.1 Simplicial Homology

In Figure 2, one can see an example of a filtered simplicial complex generated from points on a torus. As one moves from left to right, the filtration parameter, r , is increased yielding a more connected complex. In Figure 3 we show the persistence barcodes for the same shape. Note that the significant intervals corresponding to homological features that last for a long time in the filtration.



Figure 2: Example of a Lazy-Witness complex generated from randomly sampled points on a torus.

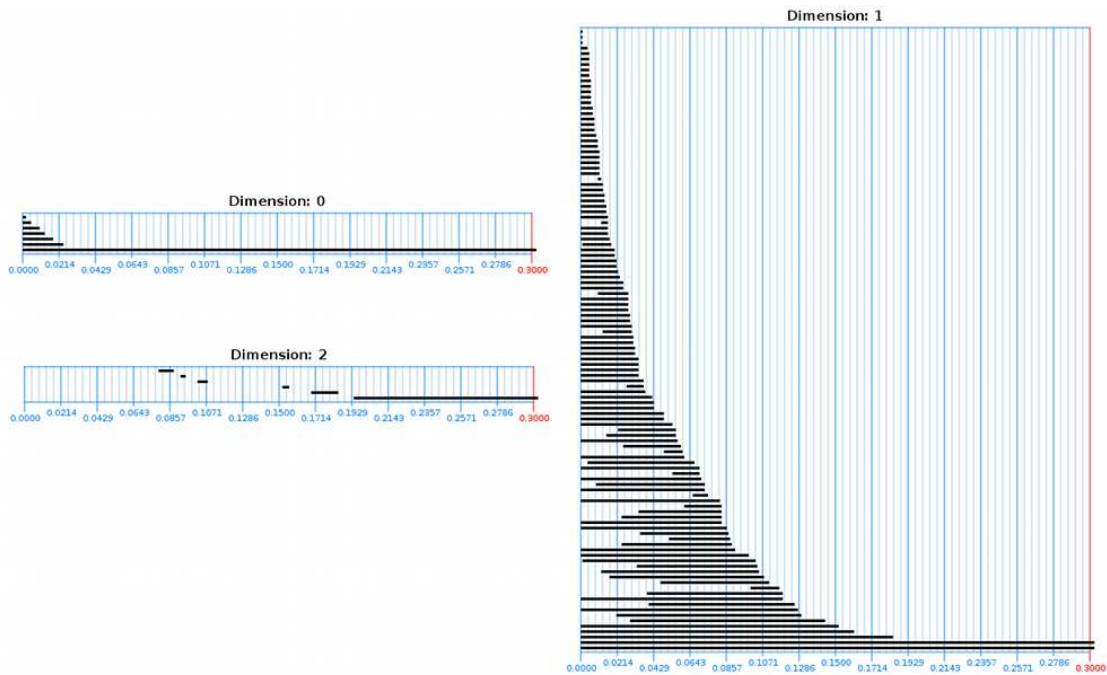


Figure 3: Persistence barcodes for a lazy-witness filtration of random points on a torus. The parameters used were: $N = 1000$, $L = 300$, $r_{\max} = 0.3$. The inner and outer radii of the torus were 0.5 and 1, respectively. The max-min selection procedure was used to create the landmark set. Note that long intervals correspond to significant homological features, and short ones are most likely the result of noise. We can see that the number of significant intervals in each dimension equals the expected Betti number.

8.2 Matlab Scripting Example - Cellular Homology

In this section we show a brief Matlab session in which the cellular homology is computed for a Klein bottle over different coefficient fields. Essentially, this code examples constructs a cellular Klein bottle, initializes persistence algorithm objects over the fields $\mathbb{Z}/2\mathbb{Z}$, $\mathbb{Z}/3\mathbb{Z}$, and \mathbb{Q} , and computes the persistence intervals. We emphasize the fact that javaPlex does not rely on Matlab in any way - it is merely being used as a scripting platform here.

```
% get the cellular sphere of the specified dimension
stream = examples.CellStreamExamples.getCellularKleinBottle();

% get cellular homology algorithm over Z/2Z
Z2_persistence = api.Plex4.getModularCellularAlgorithm(3, 2);
% get cellular homology algorithm over Z/3Z
Z3_persistence = api.Plex4.getModularCellularAlgorithm(3, 3);
% get cellular homology algorithm over Q
Q_persistence = api.Plex4.getRationalCellularAlgorithm(3);

% compute over Z/2Z - should give (1, 2, 1)
Z2_intervals = Z2_persistence.computeIntervals(stream)

% compute over Z/3Z - should give (1, 1, 0)
Z3_intervals = Z3_persistence.computeIntervals(stream)

% compute over Q - should give (1, 1, 0)
Q_intervals = Q_persistence.computeIntervals(stream)
```

The output of this example is:

```
Z2_intervals =
Dimension: 2 [0, infinity)
Dimension: 1 [0, infinity), [0, infinity)
Dimension: 0 [0, infinity)

Z3_intervals =
Dimension: 1 [0, infinity)
Dimension: 0 [0, infinity)

Q_intervals =
Dimension: 1 [0, infinity)
Dimension: 0 [0, infinity)
```

This is exactly what we expect, due to the presence of 2-torsion in the Klein bottle.

Acknowledgments. We would like to thank the reviewers for providing useful comments.

References

- [Car09] Gunnar Carlsson, *Topology and data*, Bulletin of the American Mathematical Society **46** (2009), no. 2, 255–308.
- [dSC04] Vin de Silva and Gunnar Carlsson, *Topological estimation using witness complexes*, Eurographics Symposium on Point-Based Graphics (M. Alexa and S. Rusinkiewicz, eds.), The Eurographics Association, 2004.
- [dSMVJ10] Vin de Silva, Dmitriy Morozov, and Mikael Vejdemo-Johansson, *Dualities in persistent (co)homology*, Unpublished manuscript, 2010.
- [ELZ02] Herbert Edelsbrunner, David Letscher, and Afra Zomorodian, *Topological persistence and simplification*, Discrete Comput. Geom. **28** (2002), no. 4, 511–533, Discrete and computational geometry and graph drawing (Columbia, SC, 2001). 1949898 (2003m:52019)
- [ZC05] Afra Zomorodian and Gunnar Carlsson, *Computing persistent homology*, Discrete Comput. Geom. **33** (2005), 249–274.
- [Zom10] A. Zomorodian, *Fast construction of the Vietoris-Rips complex*, Computers & Graphics **34** (2010), no. 3, 263 – 271.

Using Regina to experiment and compute with 3-manifold triangulations and normal surfaces

Benjamin A. Burton*

1 Introduction

Three-dimensional topology is a fertile ground for algorithmic problems. Prominent amongst these are *decision problems* (e.g., recognising the unknot, or testing whether two triangulated 3-manifolds are homeomorphic); *decomposition problems* (e.g., decomposing a triangulated 3-manifold into a connected sum of prime 3-manifolds); and *recognition problems* (e.g., “naming” the 3-manifold described by a given triangulation).

In three dimensions, such problems typically have highly complex and inefficient solutions—running times are often exponential or super-exponential, and implementations are often major endeavours developed over many years (if they exist at all). This is in contrast to dimension two, where many such problems are easily solved in small polynomial time, and dimensions ≥ 4 , where such problems can become undecidable [8, 19].

Regina [2, 6] is a software package for 3-manifold topologists and knot theorists. It aims to provide powerful algorithms and heuristics to assist with decision, decomposition and recognition problems; more broadly, it includes a range of facilities for the study and manipulation of 3-manifold triangulations. It offers rich support for normal surface theory, a major algorithmic framework that recurs throughout 3-manifold topology.

Regina is now 13 years old, with over 175 000 lines of source code. It is released under the GNU General Public License, and contributions from the research community are welcome. It adheres to the following broad development principles, in order of precedence:

1. *Correctness*: Having correct output is critical, particularly since one of Regina’s key applications is in computer proofs. For example, it uses arbitrary precision integer arithmetic where it cannot be proven unnecessary, and the API documentation makes thorough use of preconditions and postconditions.
2. *Generality*: Algorithms operate in the broadest possible scenarios (within reason), and do not require preconditions that cannot be easily tested. For instance, unknot recognition runs correctly for both bounded and ideal triangulations, and even when the input triangulation is not known to be a knot complement.
3. *Speed*: Because many of its algorithms run in exponential or super-exponential time, speed is crucial. Regina makes use of sophisticated algorithms and heuristics that, whilst adhering to the constraints of correctness and generality, make it practical for real topological problems.

Regina has featured in a number of topological applications. One recent example is the resolution of Thurston’s 30-year old question of whether the Weber-Seifert dodecahedral space is non-Haken [7], using a computer proof that is the cumulation of several decades of theoretical and algorithmic developments by many different authors.

2 Overview of Regina

Regina is multi-platform, and offers a drag-and-drop installer for MacOS, an MSI-based installer for Windows, and ready-made packages for several GNU/Linux distributions. It is thoroughly documented, and stores its data files in a compressed XML format. Regina provides three levels of user interface:

- a full graphical user interface, based on the Qt framework [20];
- a scripting interface based on Python, which can interact with the graphical interface or be used a stand-alone Python module;
- a programmers’ interface offering native access to Regina’s mathematical core through a C++ shared library.

*School of Mathematics and Physics, The University of Queensland, bab@maths.uq.edu.au

There are facilities to help new users learn their way around, including an illustrated users' handbook, context-sensitive “what's this?” help, and sample data files that can be opened through the *File* \rightarrow *Open Example* menu.

Regina's core strengths are in working with triangulations, normal surfaces and angle structures. It only offers basic support for hyperbolic geometries, for which the software packages SnapPea [27] and its successor SnapPy [9] are more suitable. Regina includes implementations of high-level decision and decomposition algorithms, including the only known full implementations of 3-sphere recognition and connected sum decomposition.

For the remainder of this paper we give a short outline of Regina's core features. For a more comprehensive list, see <http://regina.sourceforge.net/docs/featureset.html>.

3 Triangulations

The most basic object in Regina is a *3-manifold triangulation*. Regina does not restrict itself to simplicial complexes; instead it uses *generalised triangulations*, a more general notion that can represent a rich array of 3-manifolds using very few tetrahedra.

Specifically, a triangulation is formed from n tetrahedra by affinely identifying (or “gluing”) some or all of their $4n$ faces in pairs. A face is allowed to be identified with another face of the same tetrahedron. It is possible that several edges of a single tetrahedron may be identified together as a consequence of the face gluings, and likewise for vertices. It is common to work with *one-vertex triangulations*, in which all vertices of all tetrahedra become identified as a single point.

Figure 1 illustrates a two-tetrahedron triangulation of the real projective space \mathbb{RP}^3 . The two tetrahedra are labelled 0 and 1, and the four vertices of each tetrahedron are labelled 0, 1, 2 and 3. Faces 012 and 013 of tetrahedron 0 are joined directly to faces 012 and 013 of tetrahedron 1, creating a solid ball; then faces 023 and 123 of tetrahedron 0 are joined to faces 132 and 032 of tetrahedron 1, effectively gluing the top of the ball to the bottom of the ball with a 180° twist.

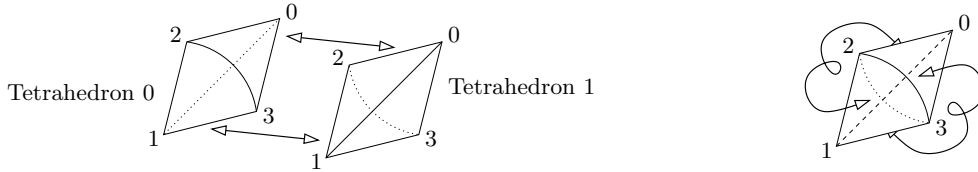


Figure 1: A triangulation of the real projective space \mathbb{RP}^3

All of this information can be encoded in a table of face gluings, which is how Regina represents triangulations:

Tetrahedron	Face 012	Face 013	Face 023	Face 123
0	1 (012)	1 (013)	1 (132)	1 (032)
1	0 (012)	0 (013)	0 (132)	0 (032)

Consider the cell in the row for tetrahedron t and the column for face abc . If this cell contains $u (xyz)$, this indicates that face abc of tetrahedron t is identified with face xyz of tetrahedron u , using the affine gluing that maps vertices a , b and c of tetrahedron t to vertices x , y and z of tetrahedron u respectively.

For any vertex V of a triangulation, the *link* of V is defined as the frontier of a small regular neighbourhood of V . It is often useful to think of vertex links as triangulated 2-dimensional surfaces, formed by inserting a small triangle into each corner of each tetrahedron, and then joining together triangles from adjacent tetrahedra along their edges. This mirrors the traditional concept of a link in a simplicial complex, but is modified to support the generalised triangulations that we use in Regina.

The triangulation above of \mathbb{RP}^3 is a *closed* triangulation, because it represents a closed 3-manifold; in a closed triangulation, every vertex link is a 2-sphere. Regina can also work with *bounded* triangulations, where one or more tetrahedron faces are not glued to anything; these unglued faces together form the boundary of the resulting 3-manifold. In a bounded triangulation, every vertex link is either a 2-sphere or a disc. The following table describes a one-tetrahedron triangulation of the solid torus $B^2 \times S^1$, whose boundary consists of two triangles (faces 023 and 123 of tetrahedron 0) that together form a 2-dimensional torus:

Tetrahedron	Face 012	Face 013	Face 023	Face 123
0	0 (301)	0 (120)		

Regina can also work with *ideal* triangulations. These are triangulations in which vertex links can be higher-genus closed surfaces (such as tori or Klein bottles). Ideal triangulations can represent non-compact 3-manifolds

(by deleting the vertices), or compact bounded 3-manifolds (by truncating the vertices). The following table shows Thurston’s famous ideal triangulation of the figure eight knot complement [24]; this triangulation has one vertex, whose link is the torus surrounding the figure eight knot in S^3 .

Tetrahedron	Face 012	Face 013	Face 023	Face 123
0	1 (210)	1 (031)	1 (231)	1 (302)
1	0 (210)	0 (031)	0 (231)	0 (302)

There are some triangulations that Regina cannot work with. It will not allow vertex links to be *bounded* surfaces other than discs (so annulus or punctured torus links are bad, for instance), and it will not allow an edge to be identified with itself in reverse. Any triangulation with such a feature will be marked as *invalid*.

Users may enter tetrahedron gluings directly; however, Regina can also create triangulations in other ways, such as importing from SnapPea [27] or other file formats, building “pre-packaged” constructions such as layered lens spaces or Seifert fibred spaces, or accessing large censuses that hold tens of thousands of triangulations of various types.

Regina can also reconstruct triangulations from *isomorphism signatures* [5]. These are short pieces of text that completely encode a triangulation; for instance, the example triangulation of \mathbb{RP}^3 above is described by the isomorphism signature `cPcbbbahh`. A feature of isomorphism signatures is that two triangulations are combinatorially isomorphic (i.e., related by a relabelling of tetrahedra and/or their vertices) if and only if their isomorphism signatures are the same. Note that, as a consequence, reconstructing a triangulation from its isomorphism signature may yield a differently-labelled (but isomorphic) copy of the original.

There are many ways to study a 3-manifold triangulation using Regina. At a high level, Regina offers decomposition and recognition routines, including: exact algorithms for 3-sphere recognition, 3-ball recognition and connected sum decomposition (these are always conclusive and correct); heuristic combinatorial algorithms for recognising much larger families of manifolds such as Seifert fibred spaces, surface bundles and graph manifolds (these “recognise” the structure of the triangulation, and will often be inconclusive); and routines from the SnapPea kernel [27] for computing volumes of hyperbolic manifolds. At a lower level, Regina can compute invariants of the underlying 3-manifold (such as homology, fundamental group and Turaev-Viro invariants), as well as combinatorial properties of the specific triangulation (such as computing the 0, 1 and 2-skeleta, or searching for common combinatorial “building blocks” within a triangulation).

Regina can also modify triangulations. Operations include local moves such as bistellar flips and edge collapses, and global operations such as barycentric subdivision, boundary coning and vertex truncation. A frequently-used operation is simplification, in which Regina uses a range of heuristic techniques to retriangulate the given 3-manifold using as few tetrahedra as it can.

4 Normal surfaces

One of Regina’s core strengths is its ability to enumerate and work with normal surfaces. A *normal surface* in a 3-manifold triangulation \mathcal{T} is a properly embedded surface in \mathcal{T} that meets each tetrahedron of \mathcal{T} in a (possibly empty) collection of disjoint curvilinear triangles and quadrilaterals, as illustrated in Figure 2.

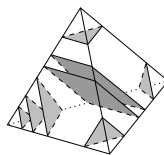


Figure 2: Normal triangles and quadrilaterals in a tetrahedron

Normal surfaces were introduced by Kneser [16] and developed by Haken for use in algorithms [11]. One of their key benefits is that they give significant insights into the structure of the underlying 3-manifold. For instance, any triangulation of \mathbb{RP}^3 (as in Figure 1) will contain a normal one-sided projective plane, and any triangulation of the solid torus (i.e., the unknot complement) will contain a normal non-separating compressing disc.

Properties such as this make normal surfaces a powerful tool for high-level recognition and decomposition routines in 3-manifold topology and knot theory. For example, they are central to algorithms for unknot recognition (where one searches for a normal disc that the unknot bounds) [11], connected sum decomposition (where one searches for normal 2-spheres that separates prime factors) [14], 3-sphere recognition [22], Hakenness testing [13], and many more.

Let \mathcal{T} be a 3-manifold triangulation with n tetrahedra. Any normal surface in \mathcal{T} can be described as an integer vector in \mathbb{R}^{7n} , whose elements count the number of triangles and quadrilaterals of each type in each tetrahedron. Specifically, for each tetrahedron Δ there are four *triangle coordinates* that count how many triangles sit within each of the four corners of Δ , and three *quadrilateral coordinates* that count how many quadrilaterals pass through Δ in each of the three possible directions. Figure 3 illustrates all seven types of triangle and quadrilateral in Δ .

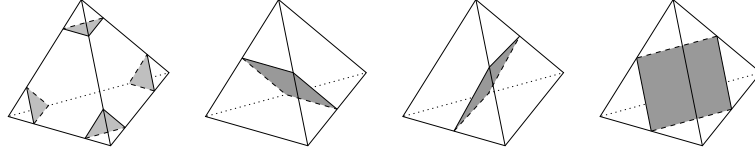


Figure 3: The seven triangle and quadrilateral types within a tetrahedron

An integer vector $\mathbf{x} \in \mathbb{R}^{7n}$ represents a normal surface in \mathcal{T} if and only if: (i) $\mathbf{x} \geq 0$; (ii) $A\mathbf{x} = 0$ for a sparse matrix A of *matching equations* derived from \mathcal{T} ; and (iii) \mathbf{x} satisfies the *quadrilateral constraints*, which require that at most one quadrilateral coordinate within each tetrahedron can be non-zero. Conditions (i) and (ii) map out a polyhedral cone in \mathbb{R}^{7n} , called the *normal surface solution cone*; condition (iii) then maps out a (typically non-convex) union of faces of this cone.

A normal surface whose vector lies on an extreme ray of the normal surface solution cone is called a *vertex normal surface*, and a normal surface whose vector lies in the Hilbert basis of this cone is called a *fundamental normal surface*. For many high-level algorithms (including all of those mentioned earlier), it can be shown that important surfaces—if they exist—can be found as vertex normal surfaces (or for some more difficult algorithms, fundamental normal surfaces). For instance, in any triangulation of a non-prime 3-manifold \mathcal{M} , there is some *vertex* normal 2-sphere that separates \mathcal{M} into a connected sum of non-trivial factors. The basic procedure then for a typical high-level algorithm is to enumerate all vertex normal surfaces (or for some problems, all fundamental normal surfaces), and then run some problem-specific test or procedure over each.

Regina comes with heavily optimised algorithms for enumerating all vertex normal surfaces [3] or fundamental normal surfaces [4] in a triangulation. It can enumerate and/or view surfaces in a number of coordinate systems, including *standard coordinates* in \mathbb{R}^{7n} (as outlined above), *quadrilateral coordinates* in \mathbb{R}^{3n} [26] (where we consider only the quadrilaterals in each tetrahedron), and *edge weight space* in \mathbb{R}^e (where we count the intersections with each of the e edges of the triangulation). It can also work with octagonal *almost normal surfaces* [23] (used in 3-sphere recognition, where we allow a single octagonal piece), and *spun normal surfaces* [25] (used with ideal triangulations, where we allow infinitely many triangles spinning in towards a vertex).

Regina offers several ways to analyse normal surfaces, both “at a glance” and in detail. It also supports the key operations of cutting a triangulation open along a normal surface and retriangulating, or crushing a surface to a point (in the Jaco-Rubinstein sense [14], where there may be additional changes in topology but which can be controlled and detected).

5 Angle structures

In addition to normal surfaces, Regina can also enumerate and analyse angle structures on a triangulation. An *angle structure* on a 3-manifold triangulation \mathcal{T} assigns non-negative internal dihedral angles to each edge of each tetrahedron of \mathcal{T} , so that (i) opposite edges of a tetrahedron are assigned the same angle; (ii) all angles in a tetrahedron sum to 2π ; and (iii) all angles around any internal edge of \mathcal{T} likewise sum to 2π (see Figure 4). Such structures are often called *semi-angle structures* [15], to distinguish them from *strict angle structures* in which all angles are strictly positive.

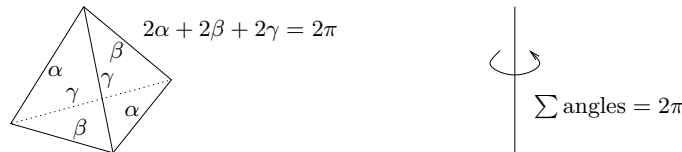


Figure 4: The conditions for an angle structure on a triangulation

Angle structures were introduced by Casson and further developed by Lackenby and Rivin [17, 21], and are a simpler (but weaker) combinatorial analogue of a complete hyperbolic structure. Some angle structures are

of particular interest: these include *taut angle structures* in which every angle is precisely 0 or π (representing “flattened” tetrahedra) [12, 18], and *veering structures* which are taut angle structures with powerful combinatorial constraints [1, 12]. Veering structures, and in some settings taut angle structures, can yield strict angle structures [12, 15], which in turn can point the way towards building a complete hyperbolic structure on \mathcal{T} [10].

Regina can construct and analyse angle structures on a 3-manifold triangulation \mathcal{T} . Specifically, conditions (i)–(iii) map out a polytope in \mathbb{R}^{3n} , where n is the number of tetrahedra; the vertices of this polytope—called *vertex angle structures*—then generate all possible angle structures on \mathcal{T} . Regina can enumerate all vertex angle structures, or (optionally) only taut angle structures, and can detect veering structures when they are present.

6 Scripting in Python

Regina offers a powerful scripting facility, whereby most of the C++ classes and functions in its mathematical engine are made available through a dedicated Python module. Python is a popular scripting language that is easy to write and easy to read, and the Python module in Regina makes it easy to quickly prototype new algorithms, run tests over large bodies of census data, or perform complex tasks that would be cumbersome through a point-and-click interface.

Users can access Regina’s Python module in two ways:

- by opening a Python console from within the graphical user interface, which allows users to directly study and/or modify data in the current working file;
- by starting the command-line program `regina-python`, which brings up a standalone Python prompt.

Users can also run their own Python scripts directly via `regina-python`, embed scripts within data files as *script packets*, or write their own libraries of frequently-used routines that will be loaded automatically each time a Regina Python session is started.

The following sample Python session constructs the triangulation of \mathbb{RP}^3 from Section 3, prints its first homology group, enumerates all vertex normal surfaces, and then locates and prints the coordinates of the vectors that represent vertex normal projective planes.

```
bab@rosemary:~$ regina-python
Regina 4.92
Software for 3-manifold topology and normal surface theory
Copyright (c) 1999-2012, The Regina development team
>>> tri = NTriangulation()
>>> t0 = tri.newTetrahedron()
>>> t1 = tri.newTetrahedron()
>>> t0.joinTo(0, t1, NPerm4(1,0,3,2))      # Glues 0 (123) -> 1 (032)
>>> t0.joinTo(1, t1, NPerm4(1,0,3,2))      # Glues 0 (023) -> 1 (132)
>>> t0.joinTo(2, t1, NPerm4(0,1,2,3))      # Glues 0 (013) -> 1 (013)
>>> t0.joinTo(3, t1, NPerm4(0,1,2,3))      # Glues 0 (012) -> 1 (012)
>>> print tri.getHomologyH1()
Z_2
>>> s = NNormalSurfaceList.enumerate(tri, NNormalSurfaceList.STANDARD, 1)
>>> print s
5 vertex normal surfaces (Standard normal (tri-quad))
>>> for i in range(s.getNumberOfSurfaces()):
...     if s.getSurface(i).getEulerCharacteristic() == 1:
...         print s.getSurface(i)
...
0 0 0 0 ; 0 1 0 || 0 0 0 0 ; 0 1 0
0 0 0 0 ; 0 0 1 || 0 0 0 0 ; 0 0 1
>>>
```

7 Future development

Regina continues to enjoy active development and regular releases. The developers are currently working towards a major version 5.0 release, which will also work with triangulated *4-manifolds* and normal hypersurfaces (much of this code is already running and well-tested in the development repository).

Users are encouraged to contribute code and offer feedback. For information on new releases, interested parties are welcome to subscribe to the low-traffic mailing list `regina-announce@lists.sourceforge.net`.

Acknowledgments. The author is grateful to many people and organisations for contributing to Regina over the years, and to Ryan Budney and William Pettersson in particular; see <http://regina.sourceforge.net/docs/credits.html> for details. The author is supported by the Australian Research Council under the Discovery Projects funding scheme (project DP1094516).

References

- [1] I. Agol. Ideal triangulations of pseudo-Anosov mapping tori. In *Topology and Geometry in Dimension Three*, volume 560 of *Contemp. Math.*, pages 1–17. Amer. Math. Soc., Providence, RI, 2011.
- [2] B. A. Burton. Introducing Regina, the 3-manifold topology software. *Experiment. Math.*, 13(3):267–272, 2004.
- [3] B. A. Burton. Optimizing the double description method for normal surface enumeration. *Math. Comp.*, 79(269):453–484, 2010.
- [4] B. A. Burton. Fundamental normal surfaces and the enumeration of Hilbert bases. Preprint, [arXiv:1111.7055](https://arxiv.org/abs/1111.7055), Nov. 2011.
- [5] B. A. Burton. Simplification paths in the Pachner graphs of closed orientable 3-manifold triangulations. Preprint, [arXiv:1110.6080](https://arxiv.org/abs/1110.6080), Oct. 2011.
- [6] B. A. Burton, R. Budney, W. Pettersson, et al. Regina: Software for 3-manifold topology and normal surface theory. <http://regina.sourceforge.net/>, 1999–2012.
- [7] B. A. Burton, J. H. Rubinstein, and S. Tillmann. The Weber-Seifert dodecahedral space is non-Haken. *Trans. Amer. Math. Soc.*, 364(2):911–932, 2012.
- [8] M. Chiodo. Finding non-trivial elements and splittings in groups. *J. Algebra*, 331:271–284, 2011.
- [9] M. Culler, N. M. Dunfield, and J. R. Weeks. SnapPy, a computer program for studying the geometry and topology of 3-manifolds. <http://snappy.computop.org/>, 1991–2011.
- [10] D. Futer and F. Guéritaud. From angled triangulations to hyperbolic structures. In *Interactions Between Hyperbolic Geometry, Quantum Topology and Number Theory*, volume 541 of *Contemp. Math.*, pages 159–182. Amer. Math. Soc., Providence, RI, 2011.
- [11] W. Haken. Theorie der Normalflächen. *Acta Math.*, 105:245–375, 1961.
- [12] C. D. Hodgson, J. H. Rubinstein, H. Segerman, and S. Tillmann. Veering triangulations admit strict angle structures. *Geom. Topol.*, 15(4):2073–2089, 2011.
- [13] W. Jaco and U. Oertel. An algorithm to decide if a 3-manifold is a Haken manifold. *Topology*, 23(2):195–209, 1984.
- [14] W. Jaco and J. H. Rubinstein. 0-efficient triangulations of 3-manifolds. *J. Differential Geom.*, 65(1):61–168, 2003.
- [15] E. Kang and J. H. Rubinstein. Ideal triangulations of 3-manifolds II; Taut and angle structures. *Algebr. Geom. Topol.*, 5:1505–1533, 2005.
- [16] H. Kneser. Geschlossene Flächen in dreidimensionalen Mannigfaltigkeiten. *Jahresbericht der Deut. Math. Verein.*, 38:248–260, 1929.
- [17] M. Lackenby. Word hyperbolic Dehn surgery. *Invent. Math.*, 140(2):243–282, 2000.
- [18] M. Lackenby. Taut ideal triangulations of 3-manifolds. *Geom. Topol.*, 4:369–395, 2000.
- [19] A. A. Markov. Insolubility of the problem of homeomorphy. In *Proc. Internat. Congress Math. 1958*, pages 300–306. Cambridge Univ. Press, New York, 1960.
- [20] Nokia Corporation. Qt – cross-platform application and UI framework. <http://qt.nokia.com/>, 1992–2012.
- [21] I. Rivin. Combinatorial optimization in geometry. *Adv. in Appl. Math.*, 31(1):242–271, 2003.
- [22] J. H. Rubinstein. An algorithm to recognize the 3-sphere. In *Proceedings of the International Congress of Mathematicians (Zürich, 1994)*, volume 1, pages 601–611. Birkhäuser, 1995.
- [23] A. Thompson. Thin position and the recognition problem for S^3 . *Math. Res. Lett.*, 1(5):613–630, 1994.
- [24] W. P. Thurston. The geometry and topology of 3-manifolds. Lecture notes, Princeton University, 1978.
- [25] S. Tillmann. Normal surfaces in topologically finite 3-manifolds. *Enseign. Math. (2)*, 54:329–380, 2008.
- [26] J. L. Tollefson. Normal surface Q -theory. *Pacific J. Math.*, 183(2):359–374, 1998.
- [27] J. R. Weeks. SnapPea: Hyperbolic 3-manifold software. <http://www.geometrygames.org/SnapPea/>, 1991–2000.

The Open Graph Drawing Framework (OGDF)

Carsten Gutwenger*

1 Introduction

We present the Open Graph Drawing Framework (OGDF), a C++ library of algorithms and data structures for graph drawing. The ultimate goal of the OGDF is to help bridge the gap between theory and practice in the field of automatic graph drawing. The library offers a wide variety of algorithms and data structures, some of them requiring complex and involved implementations, e.g., algorithms for planarity testing and planarization, or data structures for graph decomposition. A substantial part of these algorithms and data structures are building blocks of graph drawing algorithms, and the OGDF aims at providing such functionality in a reusable form, thus also providing a powerful platform for implementing new algorithms. The OGDF can be obtained from its website at <http://www.ogdf.net>. The source code is available under the GNU General Public License (GPL v2 and v3).

2 Major Design Concepts

Many sophisticated graph drawing algorithms build upon complex data structures and algorithms, thus making new implementations from scratch cumbersome and time-consuming. Obviously, graph drawing libraries can ease the implementation of new algorithms a lot. E.g., the AGD library [1], OGDF's predecessor, was very popular in the past, since it covered a wide range of graph drawing algorithms and—together with the LEDA library [23]—data structures. However, the lack of publicly available source-code restricted the portability and extendability, not to mention the understanding of the particular implementations. Other currently available graph drawing libraries suffer from similar problems, or are even only commercially available or limited to some particular graph layout methods.

Our goals for the OGDF were to transfer essential design concepts of AGD and to overcome AGD's main deficiencies for use in academic research. Our main design concepts and goals are the following:

- Provide a wide range of graph drawing algorithms that allow a user to reuse and replace particular algorithm phases by using a dedicated *module mechanism*.
- Include *sophisticated data structures* that are commonly used in graph drawing, equipped with rich public interfaces.
- A *self-contained* source code that does not require additional libraries (except for some optional LP-/ILP-based algorithms).
- *Portable* C++-code that supports the most important compilers for the major operating systems (Linux, MacOS, and Windows) and that is available under an *open source license* (GPL).

2.1 Modularization

In the OGDF, an algorithm (e.g., a graph drawing algorithm or an algorithm that can be used as building block for graph drawing algorithms) is represented as a class derived from a base class defining its interface. Such algorithm classes are also called *modules* and their base classes *module types*. E.g., general graph layout algorithms are derived from the module type `LayoutModule`, which defines as interface a `call` method whose parameters provide all the relevant information for the layout algorithm: the graph structure (`Graph`) and its graphical representation like node sizes and coordinates (`GraphAttributes`). The algorithm then obtains this information and stores the computed layout in the `GraphAttributes`.

*Departement of Computer Science, TU Dortmund, Germany, carsten.gutwenger@tu-dortmund.de

Using common interface classes for algorithms allows us to make algorithms exchangeable. We can write an implementation that utilizes several modules, but each module is used only through the interface defined by its module type. Then, we can exchange a module by a different module implementing the same module type. The OGDF provides a mechanism called module options that even makes it possible to exchange modules at runtime. Suppose an algorithm A defines a module option M of a certain type T representing a particular phase of the algorithm, and adds a set-method for this option. A *module option* is simply a pointer to an instance of type T , which is set to a useful default value in A 's constructor and called for executing this particular phase of the algorithm. Using the set-method, this implementation can be changed to *any* implementation implementing the module type T , even new implementations not contained in the OGDF itself.

Module options are the key concept for modularizing *algorithm frameworks*, thus allowing users to experiment with different implementations for particular phases of the algorithm, or to evaluate new implementations for phases without having to implement the whole framework from scratch.

2.2 Self-contained and Portable Source Code

It was important for us to create a library that runs on all important systems, and whose core part can be built without installing any further libraries. Therefore, all required basic data structures are contained in the library, and only a few modules based on linear programming require additional libraries: COIN-OR [22] as LP-solver and ABACUS [19] as branch-and-cut framework.

For reasons of portability and generality, the library provides only the drawing algorithms themselves and not any graphical display elements. Such graphical display would force us to use very system-dependent GUI or drawing frameworks, or to have the whole library based on some cross-platform toolkit. Instead of this, the OGDF simply computes basic layout information like coordinates of nodes or bend points, and an application that uses the OGDF can create the required graphical display by using the GUI framework of its choice. For creating graphics in common image formats, the OGDF project provides the command line utility `gm12pic`¹. This utility converts graph layouts stored in GML or OGML file formats into images (e.g., PNG, JPEG, EPS, PDF, SVG).

3 Algorithms and Data Structures

Apart from many basic data structures and algorithms, OGDF contains the following sophisticated algorithms and data structures.

3.1 General Graph Algorithms

Augmentation and Subgraph Algorithms. Several augmentation modules are currently available in the library for adding edges to a graph to achieve biconnectivity. This can be done either by disregarding the planarity of the graph or by taking care not to introduce non-planar subgraphs.

There are two algorithms for augmenting a planar graph to a planar biconnected graph: The module `PlanarAugmentation` implements the Fialko-Mutzel augmentation algorithm [8], which performs very good in practice, and `DfsMakeBiconnected` is a simple, DFS-based algorithm. A special variant of the planar augmentation problem is solved by the `PlanarAugmentationFix` module. Here, a planar graph with a fixed planar embedding is given, and this embedding shall be extended such that the graph becomes biconnected. `PlanarAugmentationFix` implements the optimal, linear-time algorithm by Gutwenger, Mutzel, and Zey [15].

Two modules are available to compute acyclic subgraphs of a digraph. `DfsAcyclicSubgraph` computes an acyclic subgraph in linear time by removing all back arcs in a depth-first-search tree of G . On the other hand, `GreedyCycleRemoval` implements the linear-time greedy algorithm by Eades and Lin [7]. If $G = (V, A)$ is connected and has no two-cycles, the algorithm guarantees that the number of non-feedback arcs is at least $|A|/2 - |V|/6$.

Graph Decomposition. Besides the basic algorithms for computing the biconnected components of a graph [25, 18], the OGDF provides further powerful data structures for graph decomposition. `BCTree` represents the decomposition of a graph into its biconnected components as a BC-tree and `StaticSPQRTree` represents the decomposition of a biconnected graph into its triconnected components as an SPQR-tree [5]. Both data structures can be built in linear time; the latter constructs the SPQR-tree by applying the corrected version [14] of Hopcroft

¹available at <http://www.ogdf.net/doku.php/project:gm12pic>

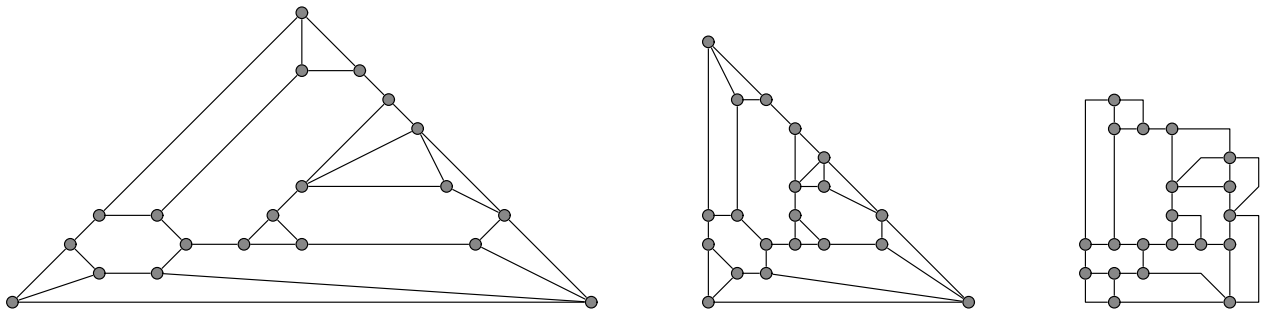


Figure 1: A triconnected planar graph drawn with `PlanarStraightLayout`, `PlanarDrawLayout`, and `MixedModelLayout` (from left to right).

and Tarjan’s algorithm [17] for decomposing a graph into its triconnected components. The OGDF is one of the few places where one can find a correct implementation of this complex and involved algorithm.

Planarity and Planarization. The OGDF provides a unique collection of algorithms for planar graphs, including algorithms for planarity testing and planar embedding, computation of planar subgraphs, and edge reinsertion. These algorithms can be combined using the planarization approach, yielding excellent crossing minimization heuristics. The planarization approach for crossing minimization is realized by the module `SubgraphPlanarizer`, and the two layout algorithms `PlanarizationLayout` and `PlanarizationGridLayout` implement a complete framework for planarization and layout.

3.2 Graph Drawing Algorithms

Graph drawing algorithms form the heart of the library. The OGDF provides flexible frameworks with interchangeable modules for various drawing paradigms, including the *planarization approach* for drawing general, non-planar graphs, the *Sugiyama framework* for drawing hierarchical graphs, and the *multilevel-mixer*, which is a general framework for multilevel, energy-based graph layout.

Planar Drawing Algorithms. The class `PlanarStraightLayout` implements planar straight-line drawing algorithms based on a shelling order of the nodes. For a graph with n nodes, the algorithm guarantees to produce a drawing on a $(2n - 4) \times (n - 2)$ grid, with convex faces if the graph is triconnected. An improved version of `PlanarStraightLayout` is `PlanarDrawLayout`. It guarantees a smaller grid size of $(n - 2) \times (n - 2)$. Some sample drawings of a triconnected graph are shown in Fig. 1.

In mixed-model layouts, each edge is drawn in an orthogonal fashion, except for a small area around its endpoints. The class `MixedModelLayout` represents the layout algorithm by Gutwenger and Mutzel [13], which is based upon ideas by Kant [21] and also uses a shelling order. This algorithm draws a d -planar graph G on a grid such that every edge has at most three bends and the minimum angle between two edges is at least $\frac{2}{d}$ radians. The grid size is at most $(2n - 6) \times (\frac{3}{2}n - \frac{7}{2})$. See Fig. 1 for an example.

Orthogonal drawings represent edges as sequences of horizontal and vertical line segments. Bends occur where these segments change directions. The OGDF provides orthogonal layout algorithms for graphs without degree restrictions; these are embedded in the planarization approach realized by `PlanarizationLayout`.

Hierarchical Drawing Algorithms. The OGDF provides a flexible implementation of Sugiyama’s framework [24] for drawing directed graphs in a hierarchical fashion; see Fig. 2. This framework basically consists of three phases, and for each phase various methods and variations have been proposed in the literature. The corresponding OGDF implementation `SugiyamaLayout` provides a module option for each of the three phases; optionally, a packing module can be used to pack multiple connected components of the graph.

Though the commonly applied approach for hierarchical graph drawing is based on the Sugiyama framework, there is a much better alternative that produces substantially less edge crossings: the *upward planarization approach* [3] adapts the crossing minimization procedure known from the planarization approach. The OGDF contains a modularized framework for upward planarization as well, however there is currently only a single implementation for each phase.

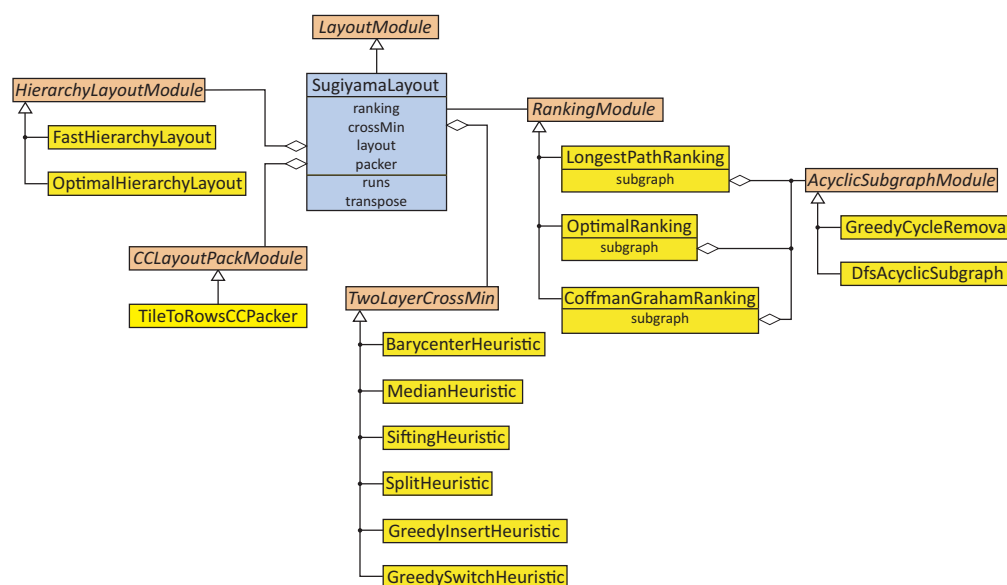


Figure 2: Sugiyama’s framework for hierarchical graph layout in the OGDF.

Energy-based Drawing Algorithms. Energy-based drawing algorithms constitute the most common drawing approach for undirected graphs. They are reasonably fast for medium sized graphs, intuitive to understand, and easy to implement—at least in their basic versions. The fundamental underlying idea of energy-based methods is to model the graph as a system of interacting objects that contribute to the overall energy of the system, such that an energy-minimized state of the system corresponds to a nice drawing of the graph. In order to achieve such an optimum, an energy or cost function is minimized. There are various models and realizations for this approach, and the flexibility in the definition of both the energy model and the objective function enables a wide range of optimization methods and applications.

The OGDF provides implementations for several classical algorithms, such as the force-directed spring embedder algorithm [6], the grid-variant of Fruchterman and Reingold [10], the simulated annealing approach by Davidson and Harel [4], the algorithm by Kamada and Kawai [20], which uses the shortest graph-theoretic distances as ideal pairwise distance values, as well as the GEM algorithm [9] and Tutte’s barycenter method [26].

In addition to these single level algorithms, the OGDF provides a generic framework for the implementation of multilevel algorithms. Multilevel approaches can help to overcome local minima and slow convergence problems of single level algorithms. Their result does not depend on the quality of an initial layout, and they are well suited also for large graphs with up to tens or even hundreds of thousands of nodes. The multilevel framework allows us to obtain results similar to those of many different multilevel layout realizations [27, 11, 16]. Instead of implementing these versions from scratch, only the main algorithmic phases—coarsening, placement, and single level layout—have to be implemented or reused from existing realizations. The module concept allows us to plug in these implementations into the framework, enabling also a comparison of different combinations as demonstrated in [2]. Fig. 3 shows two example drawings of large graphs.

On the one hand, the multilevel framework provides high flexibility for composing multilevel approaches out of a variety of realizations for the different layout steps. On the other hand, this modularity prohibits fine-tuning of specific combinations by adjusting the different phases to each other. Therefore the OGDF also contains a dedicated implementation of the fast multipole multilevel method by Hachul and Jünger [16], as well as an engineered and optimized version of this algorithm supporting multicore hardware [12].

References

- [1] D. Alberts, C. Gutwenger, P. Mutzel, and S. Näher. AGD-library: A library of algorithms for graph drawing. In *Proc. WAE ’97*, pages 112–123, 1997.
- [2] G. Bartel, C. Gutwenger, K. Klein, and P. Mutzel. An experimental evaluation of multilevel layout methods. In *Proc. Graph Drawing 2010*, number 6502 in LNCS, pages 80–91. Springer, 2010.
- [3] M. Chimani, C. Gutwenger, P. Mutzel, and H.-M. Wong. Layer-free upward crossing minimization. *ACM J. Exp. Algorithmics*, 15:Article No. 2.2, 2010.
- [4] R. Davidson and D. Harel. Drawing graphs nicely using simulated annealing. *ACM Trans. Graph.*, 15(4):301–331, 1996.

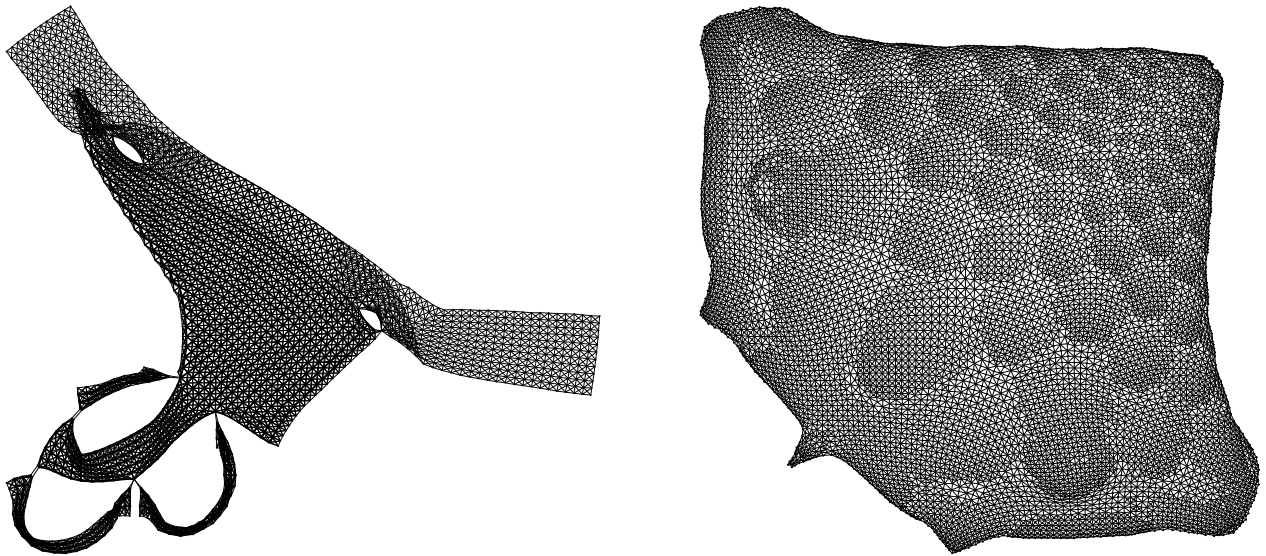


Figure 3: Two drawings obtained with OGDF's multilevel framework: graph data (left; 2,851 nodes; 15,093 edges) and graph crack (right; 10,240 nodes; 30,380 edges).

- [5] G. Di Battista and R. Tamassia. On-line maintenance of triconnected components with SPQR-trees. *Algorithmica*, 15:302–318, 1996.
- [6] P. Eades. A heuristic for graph drawing. In *Congressus Numerantium*, volume 42, pages 149–160, 1984.
- [7] P. Eades and X. Lin. A new heuristic for the feedback arc set problem. *Australian J. Combin.*, 12:15–26, 1995.
- [8] S. Fialko and P. Mutzel. A new approximation algorithm for the planar augmentation problem. In *Proc. SODA 1998*, pages 260–269, San Francisco, California, 1998. ACM Press.
- [9] A. Frick, A. Ludwig, and H. Mehldau. A fast adaptive layout algorithm for undirected graphs. In *Proc. Graph Drawing 1994*, pages 388–403, London, UK, 1995. Springer.
- [10] T. M. J. Fruchterman and E. M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.
- [11] P. Gajer and S. G. Kobourov. GRIP: Graph drawing with intelligent placement. *J. Graph Algorithms Appl.*, 6(3):203–224, 2002.
- [12] M. Gronemann. Engineering the fast-multipole-multilevel method for multicore and SIMD architectures. Master's thesis, Technische Universität Dortmund, 2009.
- [13] C. Gutwenger and P. Mutzel. Grid embedding of biconnected planar graphs. Extended Abstract, Max-Planck-Institut für Informatik, Saarbrücken, Germany, 1997.
- [14] C. Gutwenger and P. Mutzel. A linear time implementation of SPQR trees. In J. Marks, editor, *Proc. Graph Drawing 2000*, volume 1984 of *LNCS*, pages 77–90. Springer, 2001.
- [15] C. Gutwenger, P. Mutzel, and B. Zey. Planar biconnectivity augmentation with fixed embedding. In J. Fiala, J. Kratochvíl, and M. Miller, editors, *Proc. IWOCA 2009*, volume 5874 of *LNCS*, pages 289–300. Springer, 2009.
- [16] S. Hachul and M. Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In J. Pach, editor, *Proc. Graph Drawing 2004*, volume 3383 of *LNCS*, pages 285–295. Springer, 2004.
- [17] J. Hopcroft and R. E. Tarjan. Dividing a graph into triconnected components. *SIAM J. Comput.*, 2(3):135–158, 1973.
- [18] J. E. Hopcroft and R. E. Tarjan. Efficient algorithms for graph manipulation. *Communications of the ACM*, 16(6):372–378, 1973.
- [19] M. Jünger and S. Thienel. The ABACUS system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization. *Software Pract. Exper.*, 30:1325–1352, 2000. See also <http://www.informatik.uni-koeln.de/abacus/>.
- [20] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Inform. Process. Lett.*, 31(1):7–15, 1989.
- [21] G. Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16:4–32, 1996. (special issue on Graph Drawing, edited by G. Di Battista and R. Tamassia).
- [22] K. Martin. Tutorial: COIN-OR: Software for the OR community. *Interfaces*, 40(6):465–476, 2010. See also <http://www.coin-or.org>.
- [23] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [24] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Trans. Syst. Man Cybern.*, SMC-11(2):109–125, 1981.

- [25] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972.
- [26] W. T. Tutte. How to draw a graph. *Proc Lond Math Soc*, 13:743–767, 1963.
- [27] C. Walshaw. A multilevel algorithm for force-directed graph-drawing. *J. Graph Algorithms Appl.*, 7(3):253–285, 2003.

An Introduction to polymake 2.12

Michael Joswig*

Abstract

`polymake` is a tool to study the combinatorics and the geometry of convex polytopes and polyhedra. It is also capable of dealing with simplicial complexes, matroids, polyhedral fans, graphs, some objects from tropical geometry, and more.

1 Introduction

This text is meant as a short introduction to the system. It refers to features of the version 2.12, released March 19, 2012. While `polymake` is capable of dealing with a number of different mathematical objects the bulk of the code still deals with convex polytopes. Therefore we begin with polytopes, that is, convex hulls of finitely many points in Euclidean space. The idea is to show a typical use case.

The following people contributed to this version of `polymake`: Benjamin Assarf, Evgenij Gawrilow, Katrin Herr, Sven Herrmann, Lars Kastner, Benjamin Lorenz, Silke Möser, Andreas Paffenholz, Thomas Rehn, Thilo Rörig, Benjamin Schröter.

2 An Example Session

In the following commands in the `polymake` shell are preceded with a ‘>’ symbol. the language is a dialect of Perl. We start by producing a polytope which is the Cartesian product of a pentagon and a heptagon (both regular), and we want to know the *f*-vector, which records the number of faces per dimension.

```
> $p57=product(n_gon(5),n_gon(7));
> print $p57->F_VECTOR;
35 70 47 12
```

This means that our polytope called `$p57` has 35 vertices, 70 edges, 47 two-dimensional faces and twelve three-dimensional faces. The latter are the *facets* of `$p57`.

A key concept in `polymake` is that mathematical objects, such as our polytope `$p57`, have *properties* which the user is interested in. In the example above the user wants to learn about the property `F_VECTOR`. For basic usage the user can deal with objects and properties in a naive way: If a user asks for a specific property the system decides how to compute it and tells the result. Relevant intermediate information is kept and does not need to be re-computed if requested later. The list of available properties depends on the type of the object as well as on the individual user’s installation. The latter holds since `polymake` can be extended in ways which are transparent to the user.

We proceed with the example from above. By construction the polytope `$p57` has a great deal of symmetry, and this is what we want to exhibit in the next few commands. The group of *combinatorial automorphisms* is the automorphism of the face lattice. In this case it is generated by four elements, each of which is written as a pair of permutations. The first in each pair gives the action on the twelve facets, numbered from 0 to 11, while the second gives the action on the 35 vertices. Each line of the output is truncated slightly.

```
> $Aut=automorphisms($p57->VERTICES_IN_FACETS);
> print $Aut;
(<7 6 2 3 4 5 1 0 9 8 10 11> <0 6 5 ...>)
(<0 1 2 5 4 3 6 7 8 9 11 10> <0 1 2 ...>)
(<1 2 6 3 4 5 7 8 9 0 10 11> <1 2 3 ...>)
(<0 1 2 4 5 10 6 7 8 9 11 3> <7 8 9 ...>)
```

*Fachbereich Mathematik, TU Darmstadt, 64289 Darmstadt, Germany, joswig@mathematik.tu-darmstadt.de

It is a new feature of version 2.12 that, via an interface to `permlib` [2], `polymake` can treat finite permutation groups as objects.

```
> $G=new group::Group(GENERATORS=>[map { $_->first } @$Aut]);
> print $G->ORDER;
140
```

The whole `polymake` project is subdivided into *applications* which center around one (or slightly more than one) kind of object. We dealt with the application `polytope` so far, and now we do a context switch to the application for groups. This way we can access functions from that application without the need to qualify the application such as in `group::Group` above.

```
> application 'group';
> $orbits=compute_domain_orbits($G);
> print $orbits;
{0 1 2 6 7 8 9}
{3 4 5 10 11}
```

This is the orbit structure on the facets. So we got two types (namely, prisms over pentagons and prisms over heptagons). The indices refer to the implicit numbering of the facets in the object `$p57`; this can be made explicit as follows.

```
> print rows_numbered(convert_to<Float>($p57->FACETS));
0:4.0489173395223 0 0 -1 -4.38128626753482
1:1.44504186791263 0 0 1 -1.2539603376627
2:1 0 0 1.10991626417474 0
3:2.6180339887499 1 -3.07768353717525 0 0
4:1 1.23606797749979 0 0 0
5:2.6180339887499 1 3.07768353717525 0 0
6:1.44504186791263 0 0 1 1.2539603376627
7:4.0489173395223 0 0 -1 4.38128626753482
8:2.07652139657234 0 0 -2.07652139657234 1
9:2.07652139657234 0 0 -2.07652139657234 -1
10:1.37638192047117 -1.37638192047117 1 0 0
11:1.37638192047117 -1.37638192047117 -1 0 0
```

Let us create a picture. The *dual graph* of a polytope is the abstract graph whose nodes are the facets (that is, the faces of codimension one) and whose edges correspond to the faces codimension two shared by two facets. Nodes corresponding to facets in the same orbit receive the same color. `polymake` uses a spring embedding algorithm to draw this graph without making use of geometric information. This way the combinatorial symmetry is automatically reflected, as shown in Figure 1.

```
> @color = map {
    $orbits->[0]->contains($_) ? "red" : "blue"
} 0..($p57->N_FACETS-1);
> $p57->VISUAL_DUAL_GRAPH(NodeColor=>\@color);
```

3 Getting polymake

`polymake` runs in a UNIX like environment. A C++ compiler (preferably `gcc 4.6.x`) and a Perl interpreter are necessary in order to run the system. Java support is necessary for most visualization features. Notice that a C++ compiler is required even if you do not write C++ code yourself. Ubuntu 11.10 is recommended, but all other recent Linux versions should work, too. FreeBSD is supported as well. The new C++ compiler project `clang` is not supported yet but will be in the near future.

It is recommended to download the `polymake` tar ball from <http://polymake.org/doku.php/download/start> and to install it via

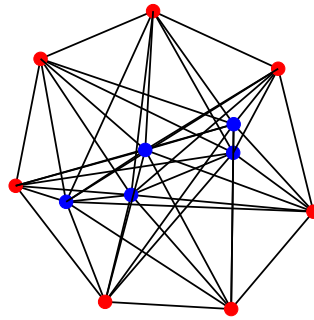


Figure 1: Dual graph of the product of a pentagon and a heptagon

```
> ./configure  
> make  
> make install
```

Installation on MacOS X from scratch is slightly more involved, hence we recommend the bundle (`polymake` 2.12 versions available for Mac OS 10.6 and 10.7).

For those with a Intel based machine but without any of the operating systems mentioned above there is a live CD version based on Linux Mint 12 LXDE (probably too large for one CD) and another smaller one based on Ubuntu Mini Remix 11.10.

Recent versions of `Cygwin` on various flavors of Windows might be able to run some portions of `polymake` but will invariably crash upon hitting the first exception (frequently used in `polymake`, and not properly supported in `Cygwin`).

References

- [1] Ewgenij Gawrilow and Michael Joswig. `polymake`: a framework for analyzing convex polytopes. In *Polytopes—combinatorics and computation (Oberwolfach, 1997)*, volume 29 of *DMV Sem.*, pages 43–73. Birkhäuser, Basel, 2000.
- [2] Thomas Rehn. `permlib` — a callable C++ library for permutation computations. <http://www.math.uni-rostock.de/~rehn/software/permlib.html>.

KINARI-Lib: A C++ library for mechanical modeling and pebble game rigidity analysis

Naomi Fox*

Filip Jagodzinski†

Ileana Streinu‡

1 Introduction

KINARI (KINematics And RIgidity, <http://kinari.cs.umass.edu>) is a software project focused on data structures and algorithms for rigidity analysis, as applied to mechanical structures, abstract sparse graphs and molecules. Designed and developed in the last author's research group at Smith College and the University of Massachusetts Amherst (LinkageLab <http://linkage.cs.umass.edu>), KINARI's first application, KINARI-Web [1], was released in 2011 as a web server for protein rigidity and flexibility analysis. An example of KINARI-Web's output, shown in Fig. 1(a), is the rigid cluster decomposition of a protein retrieved from the Protein Data Bank (PDB).

In this abstract we describe KINARI-Lib V1.0, the first public release of the underlying core library. Written in C++, KINARI-Lib implements the pebble game algorithm and provides support for body-bar-hinge and bar-joint mechanical models (illustrated in Figs. 1(b),1(c)). Links to documentation, compiled library downloads, and to the KINARI-Web server, are available from the KINARI project website.

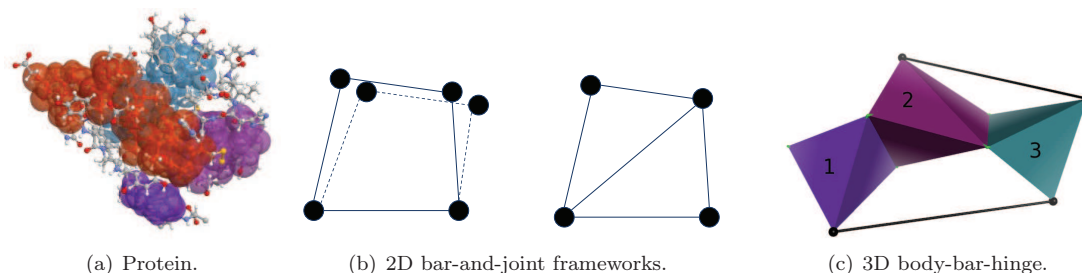


Figure 1: (a) Rigid cluster decomposition of the human insulin protein (PDB 1TRZ), as calculated by the KINARI-Web application. Only the 4 largest clusters are shown. (b) Flexible and rigid 2D bar-and-joint frameworks. (c) A 3D body-bar-hinge mechanical model.

Rigidity theory is a well-established mathematical area with applications in engineering, sensor networks, chemistry, molecular biology and CAD. It aims at finding combinatorial properties of mechanical models, or frameworks, in order to efficiently describe and compute their rigidity and flexibility properties. The two best-studied classes of frameworks, bar-and-joint and body-bar-hinge, have well-understood combinatorial properties. They are studied using associated graphs, where vertices represent rigid parts and the edges correspond to fixed-length bars whose presence removes degrees of freedom. The pebble game algorithm efficiently examines such graphs, and determines, in $O(n^2)$ -time, the total number of degrees of freedom, maximal rigid components, and over-constrained regions in a generic framework.

KINARI-Lib contains classes for the pebble game algorithm, the body-bar-hinge framework mechanical model, and special graphs on which the pebble game operates. Classes for reading from and writing to XML-formatted files are also provided for the body-bar-hinge framework and for graphs.

After some preliminaries from rigidity theory, we give an overview of the software architecture. To demonstrate its functionality, we also include two code examples for building executables: the first, for running the pebble game on graphs and the second for analyzing rigidity of body-bar-hinge frameworks.

*Department of Computer Science, University of Massachusetts, Amherst, MA, USA, fox@cs.umass.edu

†Department of Computer Science, University of Massachusetts, Amherst, MA, USA, filip@cs.umass.edu

‡Department of Computer Science, Smith College, Northampton, MA and Department of Computer Science, University of Massachusetts, Amherst, MA, USA, streinu@cs.umass.edu istreinu@smith.edu

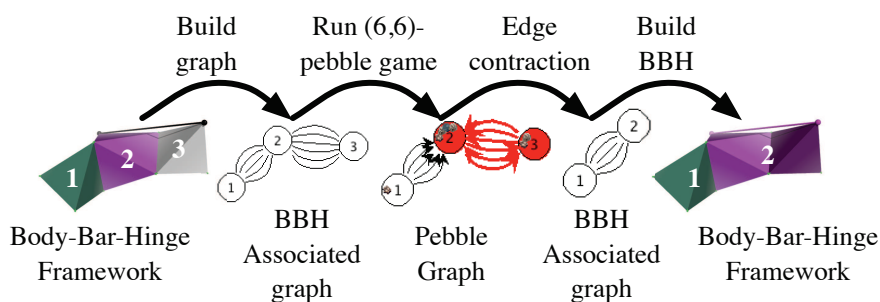


Figure 2: Rigidity analysis applied to a generic body-bar-hinge framework.

2 Background

A mechanical model, or framework, is composed of rigid pieces held together by hard constraints. The number of degrees of freedom (dofs) is the number of independent variables required to describe the position and configuration of the framework in 3D. Two well understood classes of frameworks are bar-and-joint and body-bar-hinge. A bar-and-joint framework is composed of points with distance constraints between them. Fig. 1(b) shows two bar-and-joint frameworks in the plane: one is flexible and one is rigid. A rigid framework in 2D has zero internal dofs and 3 trivial ones, which correspond to rigid motions in the plane. In 2D, the rigidity of generic bar-and-joint frameworks is completely characterized by Laman's theorem [2].

Theorem 1 Laman's Theorem 2D bar-and-joint:

A graph G with n vertices and m edges is a graph of a generically minimally rigid bar-and-joint framework iff each subgraph G' with n' vertices spans at most $m' \leq 2n' - 3$ edges, and $m = 2n - 3$.

Unfortunately, no such characterization is known in higher dimensions for bar-joint frameworks. A somewhat less general model that still lends itself well to the modeling of molecules, and which has a Laman-type characterization in 3D and higher dimensions, is the body-bar-hinge model. Such a framework is made from rigid bodies connected along hinge axes allowing only rotation around the hinge, or by fixed length bars connected at universal joints. Body-bar-hinge frameworks have associated multigraphs with one vertex for each body, 5 edges for each hinge, and 1 edge for each bar. Tay's theorem characterizes the rigidity of such generic frameworks through similar sparsity conditions on this graph [5].

Theorem 2 Tay's Theorem for 3D body-bar-hinge:

A multigraph G with n vertices and m edges is a graph of a generically minimally rigid body-bar-hinge framework iff each subgraph G' with n' vertices spans at most $m' \leq 6n' - 6$ edges, and $m = 6n - 6$.

These theorems lead to exponential-time recognition algorithms, but better methods, based on matroid partitioning and network flow, are known. Particularly convenient is the pebble game algorithm, valid for a larger class of (k, ℓ) graph sparsity conditions [3]. The algorithm starts with k pebbles placed on each vertex. Edges are considered one at a time, and accepted if they have a total of at least ℓ pebbles on their two endpoints, or when this many pebbles can be gathered through a graph search procedure. When pebbles can't be gathered, the edge is rejected. The pebbles remaining at the end represent the degrees of freedom remaining in the system.

KINARI-Lib contains an implementation of the "component pebble game algorithm", which does more than just calculating degrees of freedom: it determines the over-constraints, dofs, and rigid components (maximal subsets of vertices satisfying the sparsity condition with equality). The correctness of the pebble game calculations is guaranteed for (k, ℓ) values with $k > 0$ and $\ell \in [0, 2k)$ [3]. The values needed for applying Laman's and Tay's theorems, (2, 3) and (6, 6), fall within this range.

Tay's theorem has guarantees only for generic body-bar-hinge frameworks. The software does not check for genericity of the coordinate set, but we have provided some checks that certain combinatorial degeneracies not occur. For example, every bar endpoint must lie in one and exactly one body.

For further background on rigidity theory and the pebble game algorithm, the LinkageLab hosts an educational website (<http://linkage.cs.umass.edu/pg/>) with interactive java applets and a tutorial video [4].

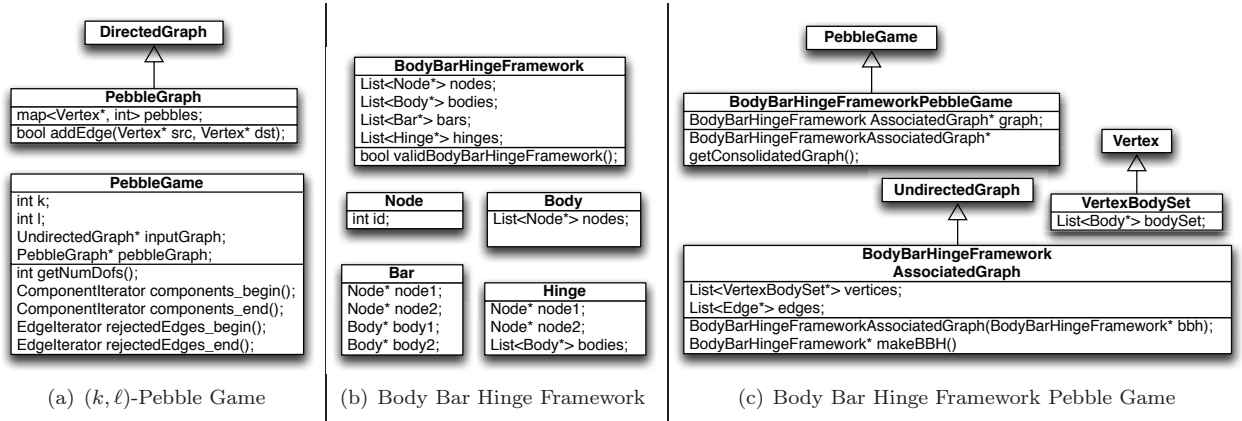


Figure 3: Overview of selected classes from KINARI-Lib.

3 KINARI-Lib software architecture

KINARI-Lib has classes for pebble games and for 3D body-bar-hinge framework mechanical modeling. To support this functionality, KINARI-Lib also provides classes for representing graphs, calculating some statistics on the body-bar-hinge frameworks, and for reading and writing each data structure in XML. A brief description of the key classes is included below and in Fig. 3. Complete documentation is distributed with the software.

PebbleGame. This class contains an implementation of the component pebble game, described in [3]. The constructor runs it by default, and requires the (k, ℓ) sparsity parameters and a (multi)graph as input. Functions are provided to retrieve the number of degrees of freedom, the maximal (rigid) components, and to identify over-constrained edges. The class contains one **PebbleGraph** object as a member variable. A **PebbleGraph** is a special extension of a directed multigraph, on which the pebble game is played.

BodyBarHingeFramework. This class contains lists of nodes, bodies, bars, and hinges. The nodes are a discrete set of points used to define the bodies, bars, and hinges. The **Node** class represents a point, which can (optionally) have coordinates associated with it. The **Body** class contains a set of nodes. The **Bar** represents the bar connection between two **Body** objects at two **Node** endpoints. A **Hinge** connects at least two bodies, at two nodes defining the hinge axis. The **BodyBarHingeFramework::validBodyBarHingeFramework()** function can be called to check a number of conditions that are described in the API documentation.

BodyBarHingeFrameworkPebbleGame. This class provides special functionality applicable to the pebble game on body-bar-hinge frameworks. The input to this is an object of the **BBHFwkAssociatedGraph** class. The **BBHFwkAssociatedGraph** a special undirected graph associated with a body-bar-hinge framework. Each vertex points to one (or more) bodies, and the edges have bars or hinges associated with them. In this way, the post-pebble-game body-bar-hinge framework can be built. The **BodyBarHingeFrameworkPebbleGame::getConsolidatedGraph()** function is provided to output an edge-contracted graph, where each component has been consolidated into a single vertex. Because this graph is of type **BBHFwkAssociatedGraph**, it contains all the information necessary to build the simplified **BodyBarHingeFramework**, where each rigid body is maximal.

4 Code examples

We include two short examples illustrating how the KINARI-Lib functionality can be incorporated into a C++ program. The first example runs the pebble game on graphs. The second one analyzes the rigidity of 3D body-bar-hinge frameworks. The library distribution includes extended versions of the two examples.

General graph pebble game. It performs the following steps:

Step 1: Read a graph from file.

Step 2: Run the pebble game on the graph, with user-specified (k, ℓ) values

Step 3: Write an XML file containing the components, dofs, and over-constraints calculated by the pebble game.

This example also demonstrates the error handling utilities inside the KINARI code. Here, the **PebbleGame** constructor will throw a **KinariException** if the (k, ℓ) values are not in the acceptable range.

```
#include "PebbleGame.h"
#include "GraphXMLFileIO.h"
#include "ComponentsXMLWriter.h"
using namespace Kinari;
int main( int argc, char **argv ) {
    // Command line parameters:
    std::string graphfilename(argv[1]); // 1. The name of the XML file containing the input graph
    int k = atoi(argv[2]); // 2. k in the (k,l) values required to configure the pebble game
    int l = atoi(argv[3]); // 3. l in the (k,l) values required to configure the pebble game
    std::string componentfilename(argv[4]); // 4. The output file to write the components to
    try {
        GraphXMLFileIO graphReader(graphfilename);
        UndirectedGraph* ugraph = graphReader.extractGraph();
        PebbleGame pg(k,l, ugraph);
        ComponentsXMLWriter::writePebGameResultsFile(pg, componentfilename);
    } catch (KinariException e) {
        std::cerr << e.toString() << std::endl; }}

```

Rigidity analysis of Body-Bar-Hinge Frameworks. The following commented code example shows the KINARI classes and syntax for analyzing a body-bar-hinge framework using the BodyBarHingeFrameworkPebbleGame class. The steps performed mirror those shown in Fig. 2.

```
#include "BBHFwkXMLFileIO.h"
#include "GraphXMLFileIO.h"
#include "BodyBarHingeFrameworkPebbleGame.h"
using namespace Kinari;
int main( int argc, char **argv ) {
    // Read a body-bar-hinge framework from an XML file
    BBHFwkXMLFileIO bbhXMLReader("bbh.xml");
    BodyBarHingeFramework* bbh = bbhXMLReader.extractBBH();
    // Create an associated graph
    BBHFrameworkAssociatedGraph bodyGraph(bbh);
    //Play pebble game and get the output graph with edge contractions
    BodyBarHingeFrameworkPebbleGame pg(&bodyGraph);
    BBHFrameworkAssociatedGraph* edgeContractedBBHGraph = pg.getConsolidatedGraph();
    GraphXMLFileIO::writeOutGraph(*edgeContractedBBHGraph, "edgeContractedBBHGraph.xml");
    // Retrieve minimized body-bar-hinge and write to file
    BodyBarHingeFramework* consolidatedBBH = edgeContractedBBHGraph->makeBBH();
    BBHFwkXMLFileIO::writeXMLToFile(*consolidatedBBH, "postPG_BBH.xml");}

```

5 Conclusion

Because it is computationally inexpensive compared with molecular simulations, rigidity analysis is a powerful tool for the study of protein flexibility. Indeed, the main application of KINARI-Lib, deployed in our publicly available server KINARI-Web, has been in this area. This release is anticipated to facilitate our library's integration into new applications.

Acknowledgment. The development of the software described in this abstract was funded by grant DMS-0714934 as part of the Joint program in mathematical biology supported by the Directorate for Mathematical and Physical Sciences of the National Science Foundation and the National Institute of General Medical Sciences of the National Institutes of Health, and by the Defense Advanced Research Projects Agency (DARPA "23 Mathematical Challenges", under "Algorithmic Origami and Biology").

References

- [1] N. Fox, F. Jagodzinski, Y. Li, and I. Streinu. KINARI-Web: a server for protein rigidity analysis. *Nucleic Acids Res.*, 39, Web Server Issue, 2011.
- [2] G. Laman. On graphs and rigidity of plane skeletal structures. *J. Eng. Math.*, 4:331–340, 1970.
- [3] A. Lee and I. Streinu. Pebble game algorithms and sparse graphs. *Discrete Math.*, 308(8):1425–1437, 2008.
- [4] A. Lee, I. Streinu, and L. Theran. Analyzing rigidity with pebble games. In *Proc. SOCG*, pages 226–227. ACM, 2008.
- [5] T.-S. Tay. Rigidity of multigraphs I: linking rigid bodies in n-space. *J. Comb. Theory B*, 36:95–112, 1984.

Solving problems with CGAL: an example using the 2D Apollonius package

Menelaos I. Karavelas^{*†}

1 Introduction

The CGAL project [3] is an open source project that aims at providing “*easy access to efficient and reliable geometric algorithms in the form of a C++ library*”, as it is stated on the project’s web site. The development of the library started in 1995 and was initially funded by two ESPRIT LTR European projects. As of November 2003 it has been an open source project, while as of its latest public release 4.0, CGAL is distributed under the LGPL/GPL v3+ licenses. CGAL currently consists of more than 500K lines of C++ code, and supports several development platforms.

One of the major goals of CGAL is to provide robust construction of geometric entities, while at the same time strive for efficiency and genericity. These goals have influenced the library’s design since the beginning of its existence. Genericity has been achieved by employing *generic programming techniques* and by following the concept/model development paradigm. Algorithms in CGAL depend on *concepts*, and at least one *model* per concept is provided by the library. Components are interchangeable as long as they satisfy the proper concept requirements, a feature that makes CGAL a sound basis for experimentation, as well as for the development of prototype or mature codes for academic and industrial uses.

The development of CGAL is characterized by the clear separation between algorithms and data structures (i.e., the combinatorial parts of a geometric algorithm) and the geometric predicates and constructions (this is the numerical part of the geometric algorithms). Predicates and constructions are encapsulated in *kernels* or *traits classes*, and they guarantee robustness by means of the Exact Geometric Computation (EGC) paradigm [10] that the library follows. Efficiency on the other hand is achieved by employing filtering techniques (for example, cf. [2]), either at the arithmetic or the geometric level.

The library is divided in four major parts: the arithmetic and algebra layer, the geometry kernels, the various packages and the support library. The arithmetic and algebra layer offers the framework for utilizing different number types, for providing polynomials, and, in more general terms, for supporting the various kernels and packages that apply to (non-)linear geometric objects. There are currently three distinct linear kernel concepts in CGAL, and at least one model for each such concept: the 2D kernel, 3D kernel and the d D kernel¹. The support library offers, among others, STL extensions of the library, interfaces between CGAL and BGL [1] and geometric objects’ generators. The bulk of the library lies, however, in its packages: there are packages for computing: arrangements, convex hulls, triangulations, Voronoi diagrams, and meshes, for performing: geometric optimization, geometry processing, and spatial searching, as well as support for: kinetic data structures and operations on cell complexes and polyhedra. The interested reader or prospective user of CGAL may consult the User and Reference Manual of the library [9].

2 2D Triangulations and Apollonius graphs in CGAL

In this abstract we focus on 2D triangulations and 2D Delaunay graphs in CGAL, and, in particular, 2D Apollonius graphs. *Voronoi diagrams* in CGAL are computed implicitly via their dual compactified *Delaunay graphs*; there is support for point and segment Euclidean Voronoi diagrams [11, 5], power diagrams [11], as well as Apollonius diagrams (a.k.a., additively-weighted Voronoi diagrams) [6]. Their dual graphs are represented via the 2D Triangulation Data Structure (TDS) [8], which can actually handle any orientable triangulated surface without boundary. The data structure has containers for triangular faces and vertices, while edges are represented implicitly. Each face has pointers to its three neighbors and vertices, and each vertex points to one of its incident faces. The user has the ability to plug-in his/her own custom vertex and face class, which are then recovered by

^{*}Department of Applied Mathematics, University of Crete, Greece, mkaravel@tem.uoc.gr

[†]Foundation for Research and Technology - Hellas, Greece

¹In fact, the models of the 2D kernel are so far also models for the 3D kernel, so at the model level, the 2D and 3D kernels are indistinguishable.

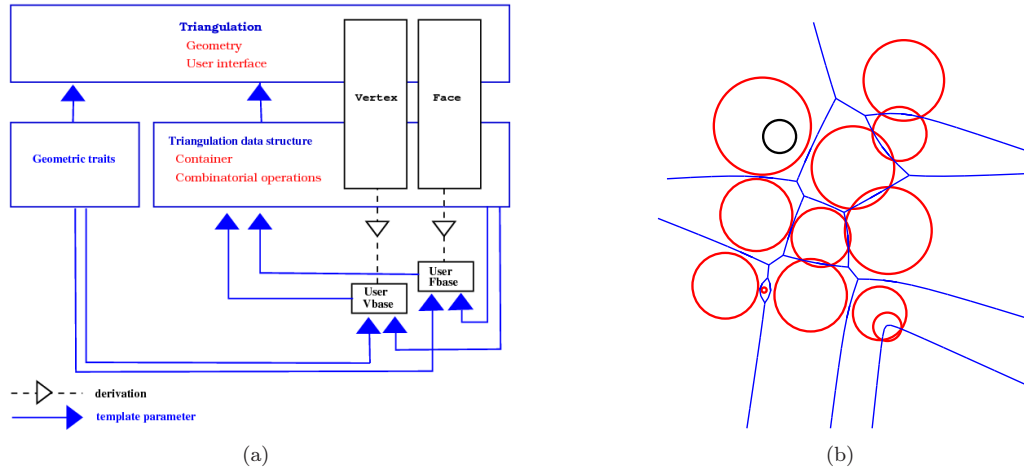


Figure 1: (a) Schematic of the design of 2D Triangulations in CGAL. (b) An instance of the 2D Apollonius diagram.

the TDS via a *rebind* mechanism. The TDS is of purely combinatorial nature; geometry is added at a higher level, where also the traits classes computing the various types of Delaunay graphs are introduced; see also Fig. 1(a) for a schematic of the design of the CGAL 2D Triangulation package. From the user's perspective, vertices and faces are seen abstractly as *handles*. The user has the ability to *traverse* the triangulation in many different ways via iterators and circulators: there are iterators for iterating over the vertices, edges or faces of the triangulation, and circulators for going around the vertices, edges and faces incident to a specific vertex. Last, but not least, point location in the Voronoi diagram, or equivalently nearest neighbor queries, is offered by all Delaunay graph classes in CGAL, while all algorithms offer incremental construction of the Delaunay graph, and, in most cases, allow for dynamic site deletion.

In the Apollonius diagram the distance of a point p from a weighted point (or site) $S = (c, w)$ is defined as $d(p, S) = \|p - c\|_2 - w$. A site can be interpreted geometrically as a disk centered at c with radius w . The Apollonius diagram has several major differences with respect to its point counterpart: sites may have empty Voronoi cells, the 1-skeleton may be disconnected (although its compactified version is connected via the site at infinity), two sites may be connected by more than one Delaunay edge, and two faces (i.e., combinatorial triangles in the Delaunay graph) may have up to two edges in common. All these characteristics are apparent in the Apollonius diagram in Fig. 1(b). The 2D Apollonius graphs in CGAL follow the same design as 2D triangulations. The user can plug-in his/her own vertex and face, has access to vertices, edges and faces via circulators and iterators, can ask for the number of the connected components of the 1-skeleton of the Apollonius diagram, and can perform nearest neighbor location queries.

3 An example application

Suppose we are given a set \mathcal{D} of n disks D_1, D_2, \dots, D_n on the plane, and we want to build a data structure that supports the following type of queries:

Given two disks D_i and D_j in \mathcal{D} , do they belong to the same connected component of the union $\cup_{k=1}^n D_k$?

Another way to pose the same problem is the following: Let $\mathcal{I}_{\mathcal{D}}$ be the intersection graph of \mathcal{D} . Given two disks in \mathcal{D} , do they belong to the same connected component of $\mathcal{I}_{\mathcal{D}}$? For simplicity, we will assume below that there is no disk in \mathcal{D} that is contained fully inside another one: the general scenario where we allow disks to be contained in other disks can also be handled, in a slightly more complicated way.

Our practical solution to this problem will be to use the 2D Apollonius graph package of CGAL [6]. Under the assumption mentioned above that there is no disk in \mathcal{D} that is contained inside another disk, the disks in \mathcal{D} correspond to vertices of the 2D Apollonius graph $AG(\mathcal{D})$ of \mathcal{D} , and, more importantly, there exists a subgraph G of $AG(\mathcal{D})$ having the same connected components as the intersection graph of \mathcal{D} (in fact, we will compute G to be a spanning forest of $\mathcal{I}_{\mathcal{D}}$). We will exploit this fact, and show how to build a mini-application on top of the 2D Apollonius graph, that also supports same-connected-component queries in $O(1)$ time. Our solution is static, in the sense that it assumes that all disks in \mathcal{D} are known in advance.

The idea is to build a spanning forest $\mathcal{F}_{\mathcal{D}}$ of $\mathcal{I}_{\mathcal{D}}$ on top of $AG(\mathcal{D})$. We will do so as follows:

1. We will modify the Apollonius graph vertex base class by adding fields for supporting a tree-like structure. This will give us an in-place implementation of the spanning forest $\mathcal{F}_{\mathcal{D}}$ of $\mathcal{I}_{\mathcal{D}}$.
2. We will create a new traits class that enriches the 2D Apollonius graph traits class with the additional predicates that are required for the computation of $\mathcal{F}_{\mathcal{D}}$.
3. We will implement the high-level class, called `Disk_intersection_subgraph_2`, that:
 - will be responsible for computing $\mathcal{F}_{\mathcal{D}}$,
 - will provide access to the connected components of $\mathcal{F}_{\mathcal{D}}$ via iterators,
 - will support the same-connected-component queries.

3.1 The vertex class

In the new vertex base class for the Apollonius graph, we are going to add three new fields: two that will be used for representing the in-place forest, understood a collection of rooted trees, and one for storing a vertex of $\mathcal{F}_{\mathcal{D}}$ that will be unique for each tree in $\mathcal{F}_{\mathcal{D}}$, and thus will act a *representative* for each tree in $\mathcal{F}_{\mathcal{D}}$ (in fact this representative node will be the root of each tree in $\mathcal{F}_{\mathcal{D}}$). The first two fields will be: (1) a vertex to the parent node in the tree of $\mathcal{F}_{\mathcal{D}}$ that our vertex belongs to, and (2) the set of children of the current vertex in the tree it belongs to. The set of children will be implemented as a `std::set`, so as to be able to efficiently determine if a vertex is a child of another vertex in some rooted tree of $\mathcal{F}_{\mathcal{D}}$. The representative vertex, which will be the third added field, will eventually be used for answering the same-connected-component query of two vertices in $\mathcal{F}_{\mathcal{D}}$, by simply comparing their representative vertices.

For implementing the vertex class of the `Disk_intersection_subgraph_2` class, named `Disk_intersection_subgraph_vertex_base_2`, we will derive from the `Apollonius_graph_vertex_base_2` class. Since the `Disk_intersection_subgraph_2` requires as traits class a superset of the traits class for computing the 2D Apollonius graph, we will pass this superset traits class as a template parameter. Last, but not least, due to the circular dependency between vertices and faces of the triangulation data structure, will also need to implement the *rebind* mechanism in order for the triangulation data structure to know the correct type of the vertex base class used.

In the code fragment below we see most of the implementation of the `Disk_intersection_subgraph_vertex_base_2` class. The implementation of some member methods is obvious and is omitted.

```
template<class Gt, bool StoreHidden = false,
        class Vb = Apollonius_graph_vertex_base_2<Gt,StoreHidden> >
class Disk_intersection_subgraph_vertex_base_2 : public Vb {
private:
    typedef Vb Base;

public:
    // public types (required by the ApolloniusGraphVertexBase_2 concept)
    typedef typename Base::Geom_traits          Geom_traits;
    typedef typename Base::Site_2               Site_2;
    typedef typename Base::Apollonius_graph_data_structure_2 Apollonius_graph_data_structure_2;
    typedef typename Base::Face_handle          Face_handle;
    typedef typename Base::Vertex_handle        Vertex_handle;

    static const bool Store_hidden = StoreHidden;

    // the rebind mechanism
    template < typename AGDS2 >
    struct Rebind_TDS {
        typedef typename Vb::template Rebind_TDS<AGDS2>::Other    Vb2;
        typedef Disk_intersection_subgraph_vertex_base_2<Gt,Store_hidden,Vb2> Other;
    };

private:
    // the comparator functor that will be used in the std::set; it uses the Compare_site_2 predicate
    // which is a new predicate (it is not provided by the model of the ApolloniusGraphTraits_2 concept)
    struct Vertex_less { ... };

    // type for the set of children nodes
    typedef std::set<Vertex_handle,Vertex_less> Children_set;
```

```

// the representative vertex, the parent vertex and the set of children
Vertex_handle rep_vertex, v_parent;
Children_set children;
public:
// type for the iterator on the childred
typedef typename Children_set::const_iterator Children_iterator;

// constructors
Disk_intersection_subgraph_vertex_base_2() : Base(), rep_vertex(), v_parent() {}
Disk_intersection_subgraph_vertex_base_2(const Site_2& p) : Base(p), rep_vertex(), v_parent() {}
Disk_intersection_subgraph_vertex_base_2(const Site_2& p, Face_handle f)
    : Base(p, f), rep_vertex(), v_parent() {}

// set/get the representative vertex
inline void representative(Vertex_handle rep) { rep_vertex = rep; }
inline Vertex_handle representative() const { return rep_vertex; }

// set/get the parent vertex
inline void parent(Vertex_handle vp) { v_parent = vp; }
inline Vertex_handle parent() const { return v_parent; }

// add a new child
inline void add_child(Vertex_handle n) { children.insert(n); }

// test if v is a child of *this vertex
inline bool has_child(Vertex_handle v) const { return children.find(v) != children.end(); }

// iterators for children
inline Children_iterator children_begin() const { return children.begin(); }
inline Children_iterator children_end() const { return children.end(); }

// the number of children
inline typename Children_set::size_type number_of_children() const { return children.size(); }

// clear the container of the child nodes
inline void clear_children_container() { children.clear(); }
};

```

3.2 Augmenting the 2D Apollonius graph traits

For computing the forest $\mathcal{F}_{\mathcal{D}}$ we need two new predicates:

1. A functor that compares two disks and returns a `Comparison_result`. This comparison functor must produce a total ordering of the sites in \mathcal{D} . It is needed for constructing the set of children in each vertex of the forest (see previous subsection).
2. A functor that takes two disks and returns `true` if the two disks intersect and `false` otherwise.

One way to produce a total ordering for the disks is to first compare their centers lexicographically, and in case of equality, their radii; we are going to call the corresponding functor `Compare_site_2` (the code for this predicate is omitted). To implement the disk intersection predicate, we simply notice that, if $D_i = (c_i, r_i)$, $c_i = (x_i, y_i)$, $i = 1, 2$ are the two input disks of the predicate, they intersect if and only if the sign of the quality $(x_1 - x_2)^2 + (y_1 - y_2)^2 - (r_1 - r_2)^2$ is not positive. We are going to call this predicate `Do_intersect_2`, and its implementation may be found below. The template parameter `Gt` here stands for the `Disk_intersection_subgraph_traits_2` class (defined below).

```

template<class Gt>
class Do_intersect_2 {
protected:
    typedef Gt Geom_traits;
    typedef typename Geom_traits::Site_2 Site_2;

// functor, taken from the CGAL kernel, that computes the squared distance of two 2D points
    typedef typename Geom_traits::Kernel::Compute_squared_distance_2 D2;

```

```

public:
    // returns true if the (closures of the) disks s and t have non-empty intersection, false otherwise
    inline bool operator()(const Site_2& s, const Site_2& t) const {
        return CGAL::compare( CGAL::square(s.weight()+t.weight()), D2()(s.point(), t.point()) ) != SMALLER;
    }
};

```

Implementing the `Disk_intersection_subgraph_traits_2` class is now straightforward. We simply need to derive from the `Apollonius_graph_traits_2<K>` class, where `K` is a model of the CGAL 2D kernel, and add the two additional predicates.

```

template<class K>
class Disk_intersection_subgraph_traits_2 : public Apollonius_graph_traits_2<K> {
    typedef Disk_intersection_subgraph_traits_2<K> Self;

    // some lines of code omitted...
public:
    // typedef for the template parameter that must be a model of the 2D CGAL kernel
    typedef K Kernel;

    // types for the two new predicates
    typedef CGAL::Do_intersect_2<Self> Do_intersect_2;
    typedef CGAL::Compare_site_2<Self> Compare_site_2;
};

```

Implementing a traits class that supports arithmetic filtering is straightforward. However, this is a purely technical issue, that goes slightly beyond the scope of this abstract.

3.3 Implementing the high-level class

Having described how to modify the vertex base class, and how to augment the `Apollonius_graph_traits_2`, so as to provide the additional predicates needed for the computation of the disk intersection subgraph, we can now implement our high-level class that is responsible for the (static) computation of the disk intersection subgraph. At this point there are quite a few design choices that may be adopted. In the code extract below, we have chosen to derive, in a protected manner, from the `Apollonius_graph_2` class, instantiated with: (i) the `Triangulation_data_structure_2` class using our own vertex base, and (ii) our augmented traits class. We also show below the constructors provided for the `Disk_intersection_subgraph_2`, as well as the trivially implemented query for determining whether two disks belong to the same connected component of the union of the set of disks passed via the iterator range in the corresponding constructor.

The bulk of the work, however, is performed in the two `compute_intersection_subgraph` methods (cf. code extract below), which implement a DFS-like search on the computed Apollonius graph (the implementation is not shown here). In the first such method, we iterate over all vertices of the Apollonius graph; for each vertex v whose representative site has not been initialized, we initialize it (by setting its representative vertex to be the vertex itself), we insert it in an empty queue Q , and we start the DFS-like search by calling the second `compute_intersection_subgraph` method using as arguments the queue Q and the vertex v . In the second method, and while the queue is not empty, we pop the first element from the queue, say v , and look at its neighbors in the Apollonius graph. If a neighbor u of v has not been initialized and the disks corresponding to u and v intersect, we add u as a child of v in the same tree that v belongs to and we set the representative vertex of u to be the same as that of v . Finally, we push u at the end of the queue. Clearly, nothing has to be done if either u has already been assigned to a tree, or if u and v do not intersect.

```

template<class Gt>
class Disk_intersection_subgraph_2 : protected Apollonius_graph_2<Gt, Triangulation_data_structure_2<
    Disk_intersection_subgraph_vertex_base_2<Gt,false>,
    Triangulation_face_base_2<Gt> > >
{
    typedef Apollonius_graph_2<Gt, Triangulation_data_structure_2<
        Disk_intersection_subgraph_vertex_base_2<Gt,false>,
        Triangulation_face_base_2<Gt> > > Base;
public:
    // type definitions omitted ...

```

```

typedef typename Base::Vertex_handle      Vertex_handle;
typedef typename Base::Geom_traits        Geom_traits;

protected:
    typedef std::queue<Vertex_handle>      Queue;

    void compute_intersection_subgraph();
    void compute_intersection_subgraph(Queue& q, Vertex_handle v_rep);

public:
    // constructors
    Disk_intersection_subgraph_2(const Geom_traits& gt = Geom_traits()) : Base(gt) {}

    template<class Input_iterator>
    Disk_intersection_subgraph_2(Input_iterator first, Input_iterator beyond,
                                const Geom_traits& gt = Geom_traits()) : Base(first, beyond, gt)
    { compute_intersection_subgraph(); }

    inline bool in_same_connected_component(Vertex_handle v1, Vertex_handle v2) const
    { return v1->representative() == v2->representative(); }

    // code omitted ...
};

```

3.4 Going one step further

In our model implementation, we also provide methods for the number of connected components, as well as for the number of disks in each connected component. We have also implemented two types of iterators: one for iterating over the connected components of the intersection subgraph, and one for iterating over the vertices per connected components. In both cases we have used the `Filter_iterator` class provided in CGAL’s STL extensions [4]. It should not be too complicated to adapt our approach above to a purely incremental one. Instead of building the spanning forest on top of the Apollonius graph, we should use the Union-Find data structure to incrementally compute the spanning subgraph. An implementation of Union-Find is already available within the Support Library of CGAL [7].

Acknowledgments. The work in this paper has been partially supported by the FP7-REGPOT-2009-1 project “Archimedes Center for Modeling, Analysis and Computation”.

References

- [1] The Boost Graph Library (BGL). <http://www.boost.org/libs/graph/>.
- [2] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Appl. Math.*, 109:25–47, 2001.
- [3] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>.
- [4] M. Hoffmann, L. Kettner, S. Pion, and R. Wein. Stl extensions for cgal. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:StlExtension.
- [5] M. Karavelas. 2D segment Delaunay graphs. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:SegmentDelaunayGraph2.
- [6] M. Karavelas and M. Yvinec. 2D Apollonius graphs (Delaunay graphs of disks). In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:ApolloniusGraph2.
- [7] L. Kettner, S. Pion, and M. Seel. Profiling tools timers, hash map, union-find, modifiers. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:ProfilingTools.
- [8] S. Pion and M. Yvinec. 2D triangulation data structure. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:TDS2.
- [9] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html.
- [10] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.
- [11] M. Yvinec. 2D triangulations. In *CGAL User and Reference Manual*. CGAL Editorial Board, 4.0 edition, 2012. http://www.cgal.org/Manual/4.0/doc_html/cgal_manual/packages.html#Pkg:Triangulation2.

M4G: Manifolds for GPUs Library

André Maximo*

Luiz Velho*

Abstract

The *Manifolds for GPUs* library — M4G — is a compendium of tools to build an atlas structure from a dense-polygon mesh to represent a surface in multiresolution in the GPU. The library is divided in two main parts: the first defines a halfedge-based mesh with stellar operators and $\sqrt{2}$ -subdivision adaptivity; the second provides a set of applications to build a regular atlas-based representation of a given mesh. Both functionalities complement other publicly available libraries, placing the M4G library on an ideal position to construct a manifold-based representation of a mesh better suitable for GPUs.

1 Introduction

Recent advances in modeling, graphics and image processing allow us to envision a system where these three areas are joined together. The many mathematical theories underpinning the discrete geometry of meshes point to a convergence between geometry and image processing, while the fast pace of graphics-hardware processing power increasingly support this convergence. The main barrier for the convergence to happen is that the leading publicly available geometric softwares, such as *MeshLab* [3] and *ParaView* [6], and libraries, such as *OpenMesh* [5] and *CGAL* [7], do not yet combine all these different concepts.

A fundamental problem in geometry processing is the one of converting a surface representation form into another. This problem occurs whenever the surface data is generated or acquired in a representation that cannot be handled by the intended application, or the representation is not the more appropriate one. Here we present a library and a set of tools to convert from the traditional surface representation, namely a triangulated two-manifold mesh, to an atlas-based representation suitable for applications in graphics processing units (GPUs). The key features of our representation are two-fold. First, it allows adaptivity control of large and complex surface models. Second, it offers a “minimum” interface that should be general enough for supporting conversion from most representation forms. Both features were made possible by the combination of two well-established notions in geometry processing: an atlas and a mesh subdivision scheme.

An *atlas*, $\mathcal{A} = \{(U_i, \varphi_i)\}_{i \in I}$, on a surface $S \subset \mathbb{R}^3$ is a collection of *charts*, (U_i, φ_i) , where U_i is a subset of S , called the *chart domain*, and $\varphi_i : U_i \rightarrow \varphi_i(U_i) \subseteq \mathbb{R}^2$ is a bijective map, called the *chart map*, that maps U_i onto a subset, $\varphi_i(U_i)$, of \mathbb{R}^2 . The chart domains “cover” S , i.e., $S = \bigcup_{i \in I} U_i$, and two or more of them may overlap at the same point in S .

From a given atlas \mathcal{A} on S , we build a polygonal mesh representation of S . Our construction is based on an adaptive subdivision scheme [22], which can be controlled by the CPU, and a dynamic tessellation, which can be done by the GPU, aiming at representing any mesh property with multiresolution. The resulting representation is a dynamic adaptive surface representation appropriate for GPU processing.

In this work we present a library dedicated to represent a discrete mesh in the computational perspective of GPUs. Our *Manifolds for GPUs* library — M4G — provides an adaptive representation of a mesh in the CPU, using charts in an atlas structure, that can be used to further tessellate regions of the same mesh in the GPU, on each chart domain. The M4G library also provides a number of useful applications to promote a GPU-based geometric system: different conversion algorithms from triangular to quadrangular (*tri2quad*) mesh based on recent articles [9, 21]; mesh simplification using CGAL [7] to create base meshes from dense-polygon meshes; fast geodesics curves computation [1, 20] for each edge of a base mesh; mesh parameterization also using CGAL [7] to build chart domains for each face of a base mesh; and a simplify procedure to create regularly sampled charts based on triangle rasterization using ordinary scan conversion from OpenGL [4]. In summary our library is a compendium of source codes and applications to aid the development of a fully integrated geometric system.

*Institute of Pure and Applied Mathematics, {andmax, lvelho}@impa.br

2 Related Work

The concept of atlas generalizes the one of parameterization, and it has been used in the context of texture mapping [15], remeshing [17], and geometry encoding [18]. In all cases, the surface S was assumed to be represented by a polygonal mesh. Here, we employ the notion of atlas without assuming any particular representation form for S . Instead, we assume that the atlas \mathcal{A} is described by a network of curves, e.g. geodesics, on S which defines the chart domain boundaries. Figure 1 details this concept in our library, a dense-polygon mesh is simplified to a base mesh and a set of geodesic curves are computed for each edge of the base mesh.

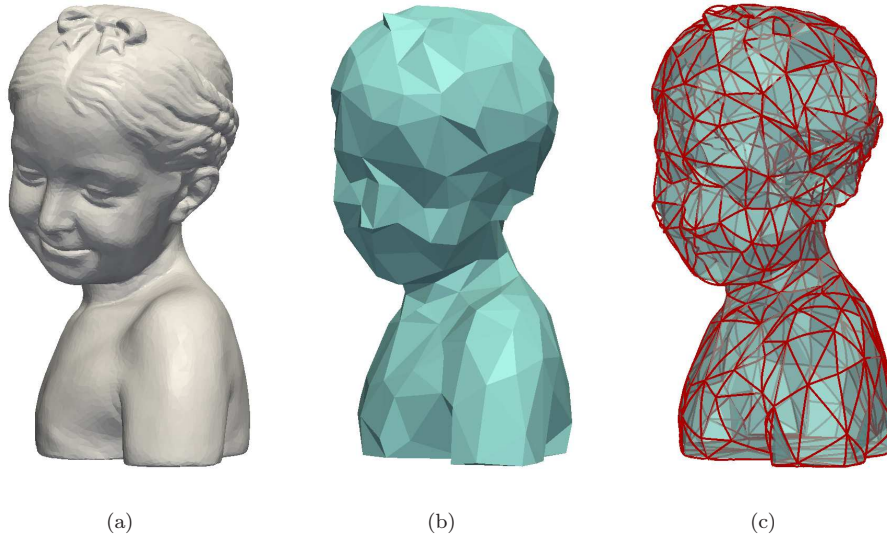


Figure 1: Example of the Bimba dataset, showing the original dense mesh (a), the base mesh (b) and the geodesics curves associated with the edges of the base mesh (c). The rendering was done using ParaView [6].

The adaptive multiresolution feature of the representation provided by our M4G library is particularly suited for progressive visualization [12]. The ability to separate the mesh into parts that must be sampled densely and parts that can be represented by coarse geometry is also useful for reducing the bandwidth required for transmitting a surface between different computational domains (e.g. hard-disk, main memory and GPU memory).

Perhaps the most powerful concept behind the manifold representation is that it enable us to work locally on a surface similarly to the two-dimensional Euclidean space. This is achieved through a parameterization, that establishes a mapping of the surface embedded in 3D to a region of the plane. The parameterization also defines a coordinate system on the surface. But since this mapping is essentially flattening a curved surface, inevitably it will cause geometric distortions. Parameterization methods try to reduce these distortions in order to preserve certain properties, such as angles, distances and areas [11]. Also, depending on how this mapping is computed, it can be designed to conform to a canonical region, such as a regular polygon (therefore constraining the boundary) or to leave the boundary free as an additional degree of freedom for distortion minimization [10]. One of the tools of our M4G library is a parameterization tool allowing the exploration of methods and border constraints to construct charts for a given mesh based on CGAL [7].

Surface parameterization also depends on the topology of the surface. Since the topology is, in general, different from the plane it is not possible to map the entire surface without cutting it open. In that respect, the parameterization methods can be classified into local and global, that respectively compute the mapping for small surface patches or the entire surface [17].

The atlas structure we consider relies on a network of curves on the surface. Based on the assumption that we can compute curves across chart domains, as well as subdivide their boundary curves, we can define an intrinsic, curve-based parametrization of each chart. This kind of parametrization is particularly useful because it naturally yields a chart map, and allows for a good control of both the map and the boundary of the region. An example of this concept is the *Multiresolution Adaptive Parameterization of Surfaces* (MAPS) [13]. In addition, it has been successfully used before for computing geodesic distances in the context of mesh parametrization [14] and remeshing [19]. Finally, the collection of individual charts, i.e. the domains and their associated maps, is a piecewise parametrization for the whole surface.

Independently of the surface representation, very often for compatibility reasons, it becomes necessary to convert to a polygonal approximation in order to perform certain tasks, such as visualization. In these situations, it is desirable to have a mechanism to produce an adaptive mesh. Examples of such strategies are the progressive meshes [12] and the stellar 4- k meshes [24]. Here we adopt the 4-8 adaptation scheme [23], which is a variant of the stellar 4- k mesh, in our base data structure inside our library.

3 The M4G Library

Our library is divided in eight repositories, publicly available in our institution gitorious server (<http://gitorious.impa.br/about>) under the GNU GPL 3 License [2]. The first repository contains our main contribution and it is called *adaptive 4-8 mesh* (*a48*) based on [25] and extending [23]. The *a48* repository (at `$ git clone git://gitorious.impa.br/a48/a48.git`) hosts an efficient implementation of a polygon mesh data structure, using half-edge [16] (cf. class `MESH_T` in figure 2); a powerful set of operators based on Stellar Theory, named stellar operators, for mesh manipulation (cf. class `STELLAR_MESH_T`); and an adaptive mesh representation with simplification and refinement strategies (cf. class `ADAPTIVE_MESH_T`). The next two repositories, called *mesh* (`/a48/mesh.git`¹) and *stellar* (`/a48/stellar.git`¹), are subsets of the first repository, hosting all codes related to `MESH_T` and `STELLAR_MESH_T`, made available for teaching purposes. The top block of figure 2 summarizes our first repository structure (for a more in-depth documentation refer to www.impa.br/~andmax/a48).

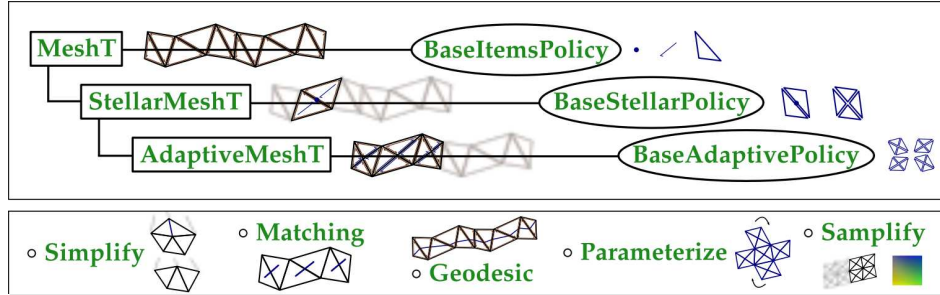


Figure 2: M4G library overview: (top) the classes and policies to manipulate and represent a meshed surface; (bottom) the applications toward a GPU-based geometric system.

The last five repositories host five different applications constructed to support a manifold-based representation of a mesh on the GPU. The first is called *simplify* (at `$ git clone git://gitorious.impa.br/m4g/simplify.git`) and contains a command-line software to expose CGAL’s simplification process [7] to build base meshes from dense-polygon meshes (cf. application `SIMPLIFY` in figure 2). The second is called *matching* (`/m4g/matching.git`¹) and uses Lemon [8] to implement a triangular-to-quadrangular application based on a 2-matching graph-based algorithm [9] on the triangular mesh dual (cf. `MATCHING`). Two other algorithms for triangular-to-quadrangular conversion [21, 25] can be used from the *tri2quad* application in the first *a48* repository. These three *tri2quad* conversion algorithms were carefully implemented and return better and correct results where `MeshLab` [3] fails.

The next repository is called *geodesic* (`/m4g/geodesic.git`¹) and contains another command-line software to expose geodesic curves computation [1] (cf. application `GEODESIC`) using the implementation made available with the *Fast exact and approximate geodesics on meshes* algorithm [20]. Each geodesic is computed in parallel (using as many CPU cores as available) over a dense mesh connecting vertices of each corresponding base-mesh edge. The fourth repository is called *parameterize* (`/m4g/parameterize.git`¹) and uses CGAL [7] to unveil several local parameterization algorithms with different border types (cf. application `PARAMETERIZE`). The last application repository is called *samplify* (`/m4g/samplify.git`¹) and uses OpenGL [4] to rasterize a surface patch generating a regularly sampled chart from a parameter domain (cf. application `SAMPLIFY`). The bottom block of figure 2 summarizes our application repositories.

The repositories of the M4G library are separate on purpose to facilitate its different usages. Each repository is a module with its own dependancies, making it easier for a subset utilization (e.g. an input triangular mesh enhanced with stellar operations and *tri2quad* functions can be used to develop a new simplification process not possible using only CGAL [7]). All eight repositories have user instructions (in a read-me file) with simple usage examples and sample models.

¹For the full repository include: `git://gitorious.impa.br` before each address.

Acknowledgments. We acknowledge the grant of the first author provided by Brazilian agency CNPq (National Council of Technological and Scientific Development). We also acknowledge the AIM@SHAPE project for the models used for testing the M4G library, such as the Bimba model.

References

- [1] Geodesic. <http://code.google.com/p/geodesic>.
- [2] GNU General Public License version 3. <http://www.gnu.org/licenses/gpl.html>.
- [3] MeshLab. <http://meshlab.sourceforge.net>.
- [4] OpenGL. <http://www.opengl.org>.
- [5] OpenMesh. <http://openmesh.org>.
- [6] ParaView. <http://www.paraview.org>.
- [7] CGAL, Computational Geometry Algorithms Library. <http://www.cgal.org>.
- [8] LEMON, Graph Library. <http://lemon.cs.elte.hu>.
- [9] J. Daniels II, M. Lizier, M. Siqueira, C. T. Silva, and L. G. Nonato. Template-based Quadrilateral Meshing. *Computers & Graphics*, 35(3):471 – 482, 2011. <http://dx.doi.org/10.1016/j.cag.2011.03.024>.
- [10] M. Desbrun. Processing Irregular Meshes. In *Proceedings of the Shape Modeling International 2002 (SMI'02)*, pages 157–, Washington, DC, USA, 2002. IEEE Computer Society. <http://portal.acm.org/citation.cfm?id=882487.884134>.
- [11] M. Floater, K. Hormann, and M. Reimers. Parameterization of Manifold Triangulations. In *Approximation Theory X: Abstract and Classical Analysis*, pages 197–209. Citeseer, 2002. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.129.899>.
- [12] H. Hoppe. Progressive meshes. In *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '96, pages 99–108, New York, NY, USA, 1996. ACM. <http://doi.acm.org/10.1145/237170.237216>.
- [13] A. Lee, W. Sweldens, P. Schröder, L. Cowsar, and D. Dobkin. MAPS: Multiresolution Adaptive Parameterization of Surfaces. In *Proc. ACM/SIGGRAPH Conf.*, pages 95–104, 1998. <http://10.1145/280814.280828>.
- [14] H. Lee, Y. Tong, and M. Desbrun. Geodesics-based One-to-One Parameterization of 3D Triangle Meshes. *IEEE MultiMedia*, 12:27–33, January 2005. <http://portal.acm.org/citation.cfm?id=1042196.1042327>.
- [15] J. Maillot, H. Yahia, and A. Verroust. Interactive Texture Mapping. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '93, pages 27–34, New York, NY, USA, 1993. ACM. <http://doi.acm.org/10.1145/166117.166120>.
- [16] M. Mäntylä. *An introduction to solid modeling*. Computer Science Press, 1988. <http://books.google.com.br/books?id=CJVRAAAAMAAJ>.
- [17] N. Ray, W. C. Li, B. Lévy, A. Sheffer, and P. Alliez. Periodic Global Parameterization. *ACM Trans. Graph.*, 25:1460–1485, October 2006. <http://doi.acm.org/10.1145/1183287.1183297>.
- [18] P. V. Sander, Z. J. Wood, S. J. Gortler, J. Snyder, and H. Hoppe. Multi-Chart Geometry Images. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 146–155, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. <http://portal.acm.org/citation.cfm?id=882370.882390>.
- [19] O. Sifri, A. Sheffer, and C. Gotsman. Geodesic-based Surface Remeshing. In *In Proc. 12th Intl. Meshing Roundtable*, pages 189–199, 2003. <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.76.1900>.
- [20] V. Surazhsky, T. Surazhsky, D. Kirsanov, S. Gortler, and H. Hoppe. Fast exact and approximate geodesics on meshes. *ACM Transactions on Graphics (TOG)*, 24(3):553–560, 2005. <http://dx.doi.org/10.1145/1186822.1073228>.
- [21] M. Tarini, N. Pietroni, P. Cignoni, D. Panozzo, and E. Puppo. Practical Quad Mesh Simplification. *Computer Graphics Forum (Special Issue of Eurographics 2010)*, 29(2):407–418, 2010. <http://vcg.isti.cnr.it/Publications/2010/TPCPP10>.
- [22] L. Velho. Stellar Subdivision Grammars. In *Proceedings of the 2003 Eurographics/ACM SIGGRAPH symposium on Geometry processing*, SGP '03, pages 188–199, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. <http://portal.acm.org/citation.cfm?id=882370.882396>.
- [23] L. Velho. A Dynamic Adaptive Mesh Library based on Stellar Operators. *Journal of Graphics Tools*, 9(2):1–29, 2004.
- [24] L. Velho and J. Gomes. Variable Resolution 4-K Meshes. In *Computer Graphics and Image Processing, 2000. Proceedings XIII Brazilian Symposium on*, pages 123–130, 2000. <http://10.1109/SIBGRA.2000.883904>.
- [25] L. Velho and D. Zorin. 4-8 Subdivision. *Computer Aided Geometric Design*, 18(5):397–427, 2001. [http://dx.doi.org/10.1016/S0167-8396\(01\)00039-5](http://dx.doi.org/10.1016/S0167-8396(01)00039-5).

GAP persistence – a computational topology package for GAP

Mikael Vejdemo-Johansson*

Abstract

We introduce a recently built package for computing persistent homology within the computer algebra system GAP. A native GAP implementation of the persistent homology algorithm allows for easier access to the group-theoretic primitives in GAP while simultaneously working on computational topology questions – enabling for instance the use of symmetry groups to study symmetric point clouds. The strongest innovation here is in the combination of persistent homology and point cloud algorithms with the group theoretic tools in GAP.

This package is different from the several existing homological algebra and simplicial topology packages already in GAP: `simpcomp`, `homalg`, `Hap`, et c., in that the focus here is on point cloud topology, and not combinatorial complexes or homological algebra.

1 Introduction

We introduce a new package for persistent homology, written entirely in native GAP. The chosen platform has several advantages over the spectrum of available package for persistent homology and topological data analysis:

- GAP has an old, strong, and productive user base within group theory and its various applications – with many uses within research into group representation theory, crystallography, et c.
- GAP is a full-scaled computational algebra package, with strong support for combinatorics and group theory. This enables future projects that use symmetry groups to draw on inherent symmetries in point clouds, and enables persistence methods for studying objects in pure mathematics.
- The package has been developed in the project HPC-GAP, and is part of an effort to produce comfortable parallelism primitives in GAP. As such, the platform choice serves as a foundation for a future implementation of the parallelism proposed by Lipsky, Skraba and Vejdemo-Johansson in [2].

This package is a genuine new addition to the computational topology landscape. The fundamental novelty is the implementation of persistent homology routines on a platform strongly optimized for computational group theoretic investigations. The platform choice is the strongest distinguishing feature – most strong persistent homology packages are not as tightly integrated in a full-scale computational algebra system; and the computational topology packages in GAP – notably `simpcomp`, `Hap`, and `homalg` are all concerned with different aspects of topology and homological algebra than our focus.

2 Current capabilities: nearest neighbours and topology

The currently available version of the package is focused on implementing methods from point cloud topology. While the capabilities of the system are ever increasing, the current implemented code is driven by immediate research requirements.

The implemented software can be divided into three main parts:

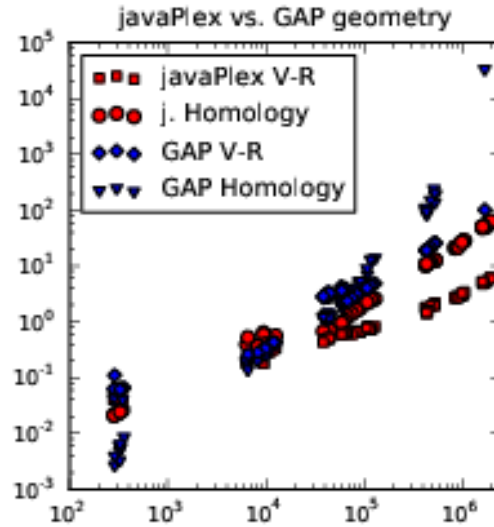
2.1 Metric geometry

GAP **persistence** comes with support for enumerating the k nearest neighbours of a point using an arbitrary metric – the Euclidean metric is provided as a helper function – and for enumerating all points in a particular ϵ -ball around a given center point, again using an arbitrary metric, though the Euclidean metric is provided.

In its current shape, the code uses brute force searches for these enumerations. While there are many much faster methods available in low ambient dimensions – using spatial trees for speeding up lookups, there do not seem to be fast methods that scale well with ambient dimension.

*School of Computer Science, University of St Andrews, mik@mcs.st-andrews.ac.uk

Figure 1: Runtimes in seconds against simplex counts for `javaPlex` and for `GAP persistence`. Up to about several hundred thousand simplices, `GAP persistence` stays about 3x-5x the time needed for `javaPlex` for comparable computations, while for a few million simplices, the runtime needed for `GAP persistence` increases over the corresponding `javaPlex` computation with up to a few orders of magnitude.



2.2 Neighbourhood graphs

The `GAP persistence` package also provides some graph theoretic primitives – capable of creating graphs with both vertex and edge decorations. These come with functionality that relies on the metric geometry above to generate neighbourhood graphs from point clouds.

Internally, given the needs of clique finding algorithms and persistent homology as a main use case, the package uses a neighbourhood list representation of graphs.

2.3 Persistent homology

The package includes a full implementation of the Incremental method for clique complex construction described by Zomorodian in [3], and a full implementation of the persistence algorithm as described by Edelsbrunner–Letscher–Zomorodian and by Zomorodian–Carlsson [1, 4], both with and without retention of representative cycles for homology classes.

3 Performance comparisons

The package is implemented entirely in native `GAP`, with the only crucial external dependency being on the high-speed hash table implementations used in the `GAP` package `orb`, included in standard `GAP` distributions. We have performed comparative benchmarks with the performance of `GAP persistence` as compared to that of `javaPlex`, and present some of these results in Figure 1. We would like to draw the reader’s attention to the fact that while an entirely interpreted language, `GAP` stays within about one order of magnitude slowdown compared to `javaPlex` until we reach about a million simplices. At this point, asymptotic complexity overtakes `GAP`’s performance, yielding a slowdown of several orders of magnitude for problems on a few million simplices.

While `GAP persistence` is slower than `javaPlex`, something that is understandable given that it is written in an entirely interpreted language, the package shows usable performance characteristics. The real power in the comparison comes in memory usage – `GAP persistence` stays very lean compared to `javaPlex`. Due to difficulties in accurately monitoring memory usage on two different platforms, both with significant internal memory management, we show no data for this – but observations during benchmarks would regularly show `javaPlex` significantly below 1G of consumed memory, even for problem sizes of several hundred thousand simplices, while `javaPlex` would comfortably use up to 4-5G for corresponding computations.

4 Usage example

See Example 4.1 for the `GAP` session described here. The language is interpreted – and the intermediate stages can be observed to get rough estimates of the computational complexity.

As an example of how to use the package, we use GAP to generate an orbit of a group action with non-trivial topology. The dihedral group D_{2k} , generated by an element c of order k and an element r of order 2, acts on \mathbb{R}^3 by rotation by $2\pi/k$ around the z -axis for the action of c and reflection through the xy -plane for the action of r .

Any non-zero point $p \in \mathbb{R}^3$ will have an orbit consisting of the vertices of a prism of a k -gon. If p is close to the xy -plane, the resulting point cloud will be close to a single circle. If p is close to the z -axis, the point cloud will be close to a pair of clusters. Otherwise, p will trace out a pair of circles – corresponding to the top and the bottom k -gons.

We do this computation for $k = 1000$, starting with the point $(1, 1, 1)$ to get two circles – each with 1000 points. We compute persistent homology up to ambient dimension, and filter the resulting barcode to highlight the interesting parts of the barcode. Since we compute with a 3-skeleton, we can expect significant noise in H_3 since the 4-simplices that could kill homology classes will not show up. We filter these classes away together with shortlived classes.

Starting instead with the point $(1, 1, 0.001)$, we get a point cloud that is almost indistinguishable from a single circle. The barcode brings up one generator each of H_0 and H_1 .

Timing prints are in milli-seconds for the entire log file.

Example 4.1 A session with GAP and the persistence package

```
Lachesis% gap.dev -b
#I Package 'nq': The executable program is not available
gap> LoadPackage("persistence");
true
gap> r := RealPart(E(1000));;
gap> i := ImaginaryPart(E(1000));;
gap> m1 := [[1,0,0],[0,1,0],[0,0,-1]];;
gap> m2 := [[r,i,0],[-i,r,0],[0,0,1]];;
gap> d2000 := Group(m1,m2);;
gap> Length(Elements(d2000)); # Size will test for finiteness first; slower.
2000
gap> cPts := Orbit(d2000, [1,1,1], OnRight);;
gap> pts := List(cPts, v -> List(v, Float));;
gap> eps := 0.01;;
gap> vr := VietorisRips(pts, eps, 3);; time;
29910
gap> int := PersistentHomology(vr, GF(3));; time;
44
gap> FilteredBarcode(int, 2, 0.005); time;
[ [ 0, 0., infinity ], [ 0, 0., infinity ], [ 1, 7.89566e-05, infinity ],
  [ 1, 7.89566e-05, infinity ] ]
2
gap> cPts := Orbit(d2000, [1,1,1/1000], OnRight);;
gap> pts := List(cPts, v -> List(v, Float));;
gap> eps := 0.01;;
gap> vr := VietorisRips(pts, eps, 3);; time;
29239
gap> int := PersistentHomology(vr, GF(3));; time;
278
gap> FilteredBarcode(int, 2, 0.005); time;
[ [ 0, 0., infinity ], [ 1, 7.89566e-05, infinity ] ]
4
```

The resulting barcode is a list of triples with the format `[dimension, birth, death]`. It is worth noticing that the datastructure `vr` is consumed during the process, and thus should be reconstructed from `vrg` and `vrc` for each renewed or halfway aborted run of the algorithm. Not surprisingly, given the input, the computation does recover the topology of the two circles.

The package comes with the capacity to load comma-separated values from a file, allowing the separation of data generation and analysis. Furthermore, one can work in the group representation theory setting of GAP and

use the command `Float` to convert vectors of rationals into vectors of machine float acceptable to the homology computation framework at the last moment. For cyclotomics, the `Float` command will be adapted in the future in all of GAP: in the meantime, we are providing a command in the library to deal with conversions from cyclotomic integers to floating point values.¹

References

- [1] H. Edelsbrunner, D. Letscher, and A. Zomorodian. Topological persistence and simplification. In *Proceedings of the 41st Annual Symposium on Foundations of Computer Science*, FOCS '00, pages 454–463, Washington, DC, USA, 2000. IEEE Computer Society. DOI: 10.1109/SFCS.2000.892133
- [2] D. Lipsky, P. Skraba, and M. Vejdemo-Johansson. A spectral sequence for parallelized persistence. arXiv:1112.1245v1 [cs.CG], December 2011.
- [3] A. Zomorodian. Technical section: Fast construction of the Vietoris-Rips complex. *Comput. Graph.*, 34(3):263–271, June 2010. DOI: 10.1016/j.cag.2010.03.007
- [4] A. Zomorodian and G. Carlsson. Computing persistent homology. In *Proceedings of the twentieth annual symposium on Computational geometry*, SCG '04, pages 347–356, New York, NY, USA, 2004. ACM. DOI: 10.1145/997817.997870.

¹Floating point arithmetic is a very new addition to GAP— and as of writing this extended abstract, the final release for version 4.5 was not yet completed.

Author Index

Adams, Henry	7
Burton, Benjamin A.	13
Fox, Naomi	29
Gutwenger, Carsten	19
Jagodzinski, Filip	29
Joswig, Michael	25
Karavelas, Menelaos I.	33
Maximo, André	39
Morozov, Dmitriy	1
Streinu, Ileana	29
Tausz, Andrew	7
Vejdemo-Johansson, Mikael	7, 43
Velho, Luiz	39

