CASC

1.0.5

Generated by Doxygen 1.9.6

1 Colored Abstract Simplicial Complex (CASC) Library 1
1.1 Getting Started . . . 1
1.1.1 Prerequisites . . . 1
1.1.2 Installing . . . 2
1.1.3 Documentation . . . 2
1.2 Versioning & Contributing . . . 2
1.3 Authors . . . 2
1.4 License . . . 2
1.5 Acknowledgments . . . 3

2 CASC License 5
2.0.1 GNU LESSER GENERAL PUBLIC LICENSE . . . 5
2.0.2 Preamble . . . 5
2.0.3 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION . . . 7
2.0.4 END OF TERMS AND CONDITIONS . . . 11
2.0.5 How to Apply These Terms to Your New Libraries . . . 11

3 Building the documentation 13
3.0.1 Documentation for Developers . . . 13

4 Frequently Asked Questions 15

5 Namespace Index 17
5.1 Namespace List . . . 17

6 Data Structure Index 19
6.1 Data Structures . . . 19

7 File Index 21
7.1 File List . . . 21

8 Namespace Documentation 23
8.1 casc Namespace Reference . . . 23
8.1.1 Typedef Documentation . . . 26
8.1.1.1 AbstractSimplicialComplex . . . 26
8.1.2 Function Documentation . . . 26
8.1.2.1 check_orientation() . . . 26
8.1.2.2 clear_orientation() . . . 27
8.1.2.3 compute_orientation() . . . 27
8.1.2.4 decimate() . . . 27
8.1.2.5 decimateBackHalf() . . . 28
8.1.2.6 decimateFirstHalf() . . . 28
8.1.2.7 edge_up() . . . 29
8.1.2.8 get() [1/3] . . . 29
8.1.2.9 get() [2/3] . . . 29

8.1.2.10 get() [3/3] . . . 30
8.1.2.11 getClosure() [1/2] . . . 30
8.1.2.12 getClosure() [2/2] . . . 31
8.1.2.13 getLink() [1/2] . . . 31
8.1.2.14 getLink() [2/2] . . . 32
8.1.2.15 getStar() [1/2] . . . 32
8.1.2.16 getStar() [2/2] . . . 32
8.1.2.17 init_orientation() . . . 33
8.1.2.18 kneighbors() [1/2] . . . 33
8.1.2.19 kneighbors() [2/2] . . . 34
8.1.2.20 kneighbors_up() [1/2] . . . 34
8.1.2.21 kneighbors_up() [2/2] . . . 35
8.1.2.22 neighbors() [1/2] . . . 35
8.1.2.23 neighbors() [2/2] . . . 36
8.1.2.24 neighbors_up() [1/2] . . . 36
8.1.2.25 neighbors_up() [2/2] . . . 37
8.1.2.26 operator"!=() . . . 37
8.1.2.27 operator==() . . . 38
8.1.2.28 perform_insertion() . . . 38
8.1.2.29 perform_removal() . . . 39
8.1.2.30 run_user_callback() . . . 39
8.1.2.31 set_difference() . . . 39
8.1.2.32 set_intersection() . . . 40
8.1.2.33 set_union() . . . 40
8.1.2.34 to_string() . . . 41
8.1.2.35 visit_BFS_down() . . . 41
8.1.2.36 visit_BFS_up() . . . 42
8.1.2.37 writeDOT() . . . 42
8.2 index_tracker Namespace Reference . . . 42
8.3 index_tracker::index_tracker_detail Namespace Reference . . . 43
8.4 util Namespace Reference . . . 44
8.4.1 Function Documentation . . . 45
8.4.1.1 int_for_each() . . . 45
8.4.1.2 make_range() [1/2] . . . 46
8.4.1.3 make_range() [2/2] . . . 46

9 Data Structure Documentation **49**
9.1 index_tracker::index_tracker_detail::BTreeNode< _T, _d > Struct Template Reference . . . 49
9.1.1 Detailed Description . . . 49
9.2 casc::simplicial_complex< traits >::EdgeID< k > Struct Template Reference . . . 50
9.2.1 Detailed Description . . . 51
9.2.2 Constructor & Destructor Documentation . . . 51

9.2.2.1 EdgeID() [1/2] . . . 51
9.2.2.2 EdgeID() [2/2] . . . 52
9.2.3 Member Function Documentation . . . 52
9.2.3.1 down() . . . 52
9.2.3.2 up() . . . 52
9.3 index_tracker::index_tracker< _T, _d > Class Template Reference . . . 53
9.3.1 Detailed Description . . . 53
9.3.2 Constructor & Destructor Documentation . . . 54
9.3.2.1 index_tracker() . . . 54
9.4 util::int_type_map< IntegerType, OutHolder, IntegerSequence, F > Struct Template Reference . . . 54
9.4.1 Detailed Description . . . 54
9.5 index_tracker::index_tracker_detail::Interval< T > Struct Template Reference . . . 55
9.5.1 Detailed Description . . . 55
9.5.2 Member Function Documentation . . . 55
9.5.2.1 operator=() . . . 56
9.6 casc::Orientable Struct Reference . . . 56
9.7 util::range< T > Struct Template Reference . . . 56
9.7.1 Detailed Description . . . 57
9.7.2 Constructor & Destructor Documentation . . . 57
9.7.2.1 range() [1/2] . . . 57
9.7.2.2 range() [2/2] . . . 57
9.7.3 Member Function Documentation . . . 58
9.7.3.1 begin() . . . 58
9.7.3.2 end() . . . 58
9.8 util::remove_first_val< Integer, IntegerSequence > Struct Template Reference . . . 58
9.8.1 Detailed Description . . . 58
9.9 util::remove_first_val< Integer, InHolder< Integer, I, Is... > > Struct Template Reference . . . 59
9.9.1 Detailed Description . . . 59
9.10 util::reverse_sequence< Integer, IntegerSequence > Struct Template Reference . . . 59
9.10.1 Detailed Description . . . 60
9.11 casc::simplicial_complex< traits >::SimplexID< k > Struct Template Reference . . . 60
9.11.1 Detailed Description . . . 62
9.11.2 Constructor & Destructor Documentation . . . 62
9.11.2.1 SimplexID() [1/2] . . . 62
9.11.2.2 SimplexID() [2/2] . . . 62
9.11.3 Member Function Documentation . . . 62
9.11.3.1 cover() . . . 63
9.11.3.2 cover_insert() . . . 63
9.11.3.3 get_simplex_up() [1/3] . . . 63
9.11.3.4 get_simplex_up() [2/3] . . . 64
9.11.3.5 get_simplex_up() [3/3] . . . 64
9.11.3.6 indices() . . . 65

9.11.4 Friends And Related Function Documentation . . . 65
9.11.4.1 operator<< . . . 65
9.12 casc::SimplexMap< Complex > Struct Template Reference . . . 65
9.12.1 Detailed Description . . . 66
9.12.2 Member Function Documentation . . . 66
9.12.2.1 get() [1/2] . . . 66
9.12.2.2 get() [2/2] . . . 67
9.12.3 Friends And Related Function Documentation . . . 67
9.12.3.1 operator<< . . . 67
9.13 casc::SimplexSet< Complex > Struct Template Reference . . . 68
9.13.1 Detailed Description . . . 69
9.13.2 Member Function Documentation . . . 69
9.13.2.1 begin() . . . 70
9.13.2.2 cbegin() . . . 70
9.13.2.3 cend() . . . 70
9.13.2.4 empty() . . . 71
9.13.2.5 end() . . . 71
9.13.2.6 erase() [1/2] . . . 71
9.13.2.7 erase() [2/2] . . . 71
9.13.2.8 find() [1/2] . . . 72
9.13.2.9 find() [2/2] . . . 72
9.13.2.10 insert() [1/2] . . . 73
9.13.2.11 insert() [2/2] . . . 73
9.13.2.12 size() . . . 73
9.13.3 Friends And Related Function Documentation . . . 74
9.13.3.1 operator<< . . . 74
9.14 casc::simplicial_complex< traits > Class Template Reference . . . 74
9.14.1 Detailed Description . . . 78
9.14.2 Member Typedef Documentation . . . 78
9.14.2.1 EdgeData . . . 79
9.14.2.2 NodeData . . . 79
9.14.3 Constructor & Destructor Documentation . . . 79
9.14.3.1 ~simplicial_complex() . . . 79
9.14.4 Member Function Documentation . . . 79
9.14.4.1 add_vertex() [1/2] . . . 79
9.14.4.2 add_vertex() [2/2] . . . 80
9.14.4.3 down() [1/3] . . . 80
9.14.4.4 down() [2/3] . . . 80
9.14.4.5 down() [3/3] . . . 81
9.14.4.6 eq() [1/2] . . . 81
9.14.4.7 eq() [2/2] . . . 82
9.14.4.8 exists() . . . 82

9.14.4.9 get_cover() [1/2] . . . 82
9.14.4.10 get_cover() [2/2] . . . 83
9.14.4.11 get_cover_insert() . . . 83
9.14.4.12 get_edge_down() [1/2] . . . 84
9.14.4.13 get_edge_down() [2/2] . . . 84
9.14.4.14 get_edge_up() [1/2] . . . 85
9.14.4.15 get_edge_up() [2/2] . . . 85
9.14.4.16 get_level() [1/2] . . . 86
9.14.4.17 get_level() [2/2] . . . 86
9.14.4.18 get_level_id() [1/2] . . . 86
9.14.4.19 get_level_id() [2/2] . . . 87
9.14.4.20 get_name() [1/3] . . . 87
9.14.4.21 get_name() [2/3] . . . 87
9.14.4.22 get_name() [3/3] . . . 88
9.14.4.23 get_simplex_down() [1/3] . . . 88
9.14.4.24 get_simplex_down() [2/3] . . . 89
9.14.4.25 get_simplex_down() [3/3] . . . 89
9.14.4.26 get_simplex_up() [1/4] . . . 90
9.14.4.27 get_simplex_up() [2/4] . . . 90
9.14.4.28 get_simplex_up() [3/4] . . . 90
9.14.4.29 get_simplex_up() [4/4] . . . 91
9.14.4.30 insert() [1/4] . . . 91
9.14.4.31 insert() [2/4] . . . 92
9.14.4.32 insert() [3/4] . . . 92
9.14.4.33 insert() [4/4] . . . 92
9.14.4.34 leq() . . . 93
9.14.4.35 lt() . . . 93
9.14.4.36 nearBoundary() . . . 94
9.14.4.37 onBoundary() . . . 94
9.14.4.38 remove() [1/3] . . . 95
9.14.4.39 remove() [2/3] . . . 95
9.14.4.40 remove() [3/3] . . . 96
9.14.4.41 size() . . . 96
9.14.4.42 up() [1/3] . . . 96
9.14.4.43 up() [2/3] . . . 97
9.14.4.44 up() [3/3] . . . 97
9.14.5 Friends And Related Function Documentation . . . 98
9.14.5.1 EdgeID . . . 98
9.14.5.2 SimplexID . . . 98
9.15 util::type_get< k, T > Struct Template Reference . . . 98
9.15.1 Detailed Description . . . 98
9.16 util::type_get< 0, type_holder< Ts... > > Struct Template Reference . . . 99

9.16.1 Detailed Description . . . 99
9.17 util::type_get< k, type_holder< Ts... > > Struct Template Reference . . . 99
9.17.1 Detailed Description . . . 99
9.18 util::type_holder< Ts > Struct Template Reference . . . 100
9.18.1 Detailed Description . . . 100
9.19 util::type_holder< T, Ts... > Struct Template Reference . . . 100
9.19.1 Detailed Description . . . 101
9.20 util::type_map< C, V > Struct Template Reference . . . 101
9.20.1 Detailed Description . . . 101
9.21 util::type_swap< TUPLE, HOLDER_FULL > Struct Template Reference . . . 102
9.21.1 Detailed Description . . . 102
9.22 util::type_swap< TUPLE, HOLDER< Ts... > > Struct Template Reference . . . 102
9.22.1 Detailed Description . . . 102

10 File Documentation **105**

10.1 include/casc/CASCFunctions.h File Reference . . . 105
10.2 CASCFunctions.h . . . 106
10.3 include/casc/CASCTraversals.h File Reference . . . 111
10.4 CASCTraversals.h . . . 113
10.5 include/casc/decimate.h File Reference . . . 122
10.6 decimate.h . . . 123
10.7 include/casc/index_tracker.h File Reference . . . 130
10.8 index_tracker.h . . . 132
10.9 include/casc/Orientable.h File Reference . . . 143
10.10 Orientable.h . . . 144
10.11 include/casc/SimplexMap.h File Reference . . . 147
10.12 SimplexMap.h . . . 148
10.13 include/casc/SimplexSet.h File Reference . . . 150
10.14 SimplexSet.h . . . 151
10.15 include/casc/SimplicialComplex.h File Reference . . . 158
10.16 SimplicialComplex.h . . . 160
10.17 include/casc/stringutil.h File Reference . . . 192
10.18 stringutil.h . . . 192
10.19 include/casc/typetraits.h File Reference . . . 193
10.19.1 Detailed Description . . . 193
10.19.2 Function Documentation . . . 194
10.19.2.1 type_name() . . . 194
10.20 typetraits.h . . . 194
10.21 include/casc/util.h File Reference . . . 195
10.22 util.h . . . 196

Index **203**

Chapter 1

Colored Abstract Simplicial Complex (CASC) Library

Master CI: Development CI:

CASC is a modern and header-only C++ library which provides a data structure to represent arbitrary dimension abstract simplicial complexes with user-defined classes stored directly on the simplices at each dimension. This is achieved by taking advantage of the combinatorial nature of simplicial complexes and new C++ code features such as: variadic templates and automatic function return type deduction. Essentially CASC stores the full topology of the complex according to a `Hasse diagram`. The representation of the topology is decoupled from interactions of user data through the use of metatemplate programming.

1.1 Getting Started

These instructions will get you a copy of the project up and running on your local machine for development and testing purposes.

1.1.1 Prerequisites

CASC does not have any dependencies other than the C++ standard library. If you wish to use CASC, you can use the header files right away. There is no binary library to link to, and no configured header file. CASC is a pure template library defined in the headers.

We use the CMake build system (version 3+), but only to build the documentation and unit-tests, and to automate installation.

Doxygen and Graphviz is used to generate the documentation.

To use CASC in your software all you will need is a working C++ compiler with full C++14 support. This includes:

- GCC Versions 5+
- Clang Versions 3, 5+[†]
- XCode 8+[†]

[†] Note that there is a known issue with Clang 4.x.x versioned compilers (including XCode version 9.[0-2]), where the most specialized unique specialization is not selected leading to a compiler error. The current workaround to this problem is to either use GCC or to obtain Clang version 5+ (XCode version 9.3beta+).

1.1.2 Installing

CASC is header only meaning that there is nothing to compile out of the box. To use CASC, simply copy the desired headers into your project and included as necessary. If you wish to install CASC using CMake to your system, even though the library is header only, you must first create a new folder to prevent in-source "builds".

```
mkdir build
cd build
```

Subsequently run CMake specifying the installation prefix and the path to the root level `CMakeLists.txt` file.

```
cmake -DCMAKE_INSTALL_PREFIX=/usr/local/ ..
make install
```

Unit tests are also packaged along with CASC and are dependent upon `Googles C++ test framework`. If you wish to build and run the tests, set the flag `-DBUILD_CASCTESTS=on` in your CMake command. CMake will then download and build `googletest` and link it with the CASC unit tests.

```
cmake -DBUILD_CASCTESTS=on ..
make
make tests          # Run tests through make
./bin/casctests     # Alternatively run the tests directly (more verbose)
```

Additional examples provided with CASC can be built in a similar fashion by passing the `-DBUILD_CASCEXAMPLES=on` flag to CMake.

1.1.3 Documentation

A current version of the documentation is available online via `github pages`. You can also build the documentation locally if you have Doxygen and Graphviz on your system. CMake will automatically try to find a working Doxygen installation. If Doxygen is found then the documentation can be built using `make casc_doc`. Otherwise CMake will report that it could not find Doxygen.

1.2 Versioning & Contributing

We use `Github` for versioning. For the versions available, please see the `releases`. If you find a bug or with to request additional functionality please file an issue in the `CASC Github project`.

1.3 Authors

John Moody
Department of Mathematics
University of California, San Diego

Christopher Lee
Department of Chemistry & Biochemistry
University of California, San Diego

See also the list of `contributors` who participated in this project.

1.4 License

1.5 Acknowledgments

This project is supported by the National Institutes of Health under grant numbers P41-GM103426 (NBCR), T32-GM008326, and R01-GM31749. It is also supported in part by the National Science Foundation under awards DMS-CM1620366 and DMS-FRG1262982.

Chapter 2

CASC License

The CASC library is free software distributed under the GNU Lesser General Public License Version 2.1. You can redistribute it and/or modify it under the terms of the LGPLv2.1 as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version. A copy of the GNU LGPLv2.1 is reproduced below.

2.0.1 GNU LESSER GENERAL PUBLIC LICENSE

Version 2.1, February 1999

```
Copyright (C) 1991, 1999 Free Software Foundation, Inc.
51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA

Everyone is permitted to copy and distribute verbatim copies
of this license document, but changing it is not allowed.

[This is the first released version of the Lesser GPL.  It also counts
 as the successor of the GNU Library Public License, version 2, hence
 the version number 2.1.]
```

2.0.2 Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public Licenses are intended to guarantee your freedom to share and change free software–to make sure the software is free for all its users.

This license, the Lesser General Public License, applies to some specially designated software packages–typically libraries–of the Free Software Foundation and other authors who decide to use it. You can use it too, but we suggest you first think carefully about whether this license or the ordinary General Public License is the better strategy to use in any particular case, based on the explanations below.

When we speak of free software, we are referring to freedom of use, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish); that you receive source code or can get it if you want it; that you can change the software and use pieces of it in new free programs; and that you are informed that you can do these things.

To protect your rights, we need to make restrictions that forbid distributors to deny you these rights or to ask you to surrender these rights. These restrictions translate to certain responsibilities for you if you distribute copies of the library or if you modify it.

For example, if you distribute copies of the library, whether gratis or for a fee, you must give the recipients all the rights that we gave you. You must make sure that they, too, receive or can get the source code. If you link other code with the library, you must provide complete object files to the recipients, so that they can relink them with the library after making changes to the library and recompiling it. And you must show them these terms so they know their rights.

We protect your rights with a two-step method: (1) we copyright the library, and (2) we offer you this license, which gives you legal permission to copy, distribute and/or modify the library.

To protect each distributor, we want to make it very clear that there is no warranty for the free library. Also, if the library is modified by someone else and passed on, the recipients should know that what they have is not the original version, so that the original author's reputation will not be affected by problems that might be introduced by others.

Finally, software patents pose a constant threat to the existence of any free program. We wish to make sure that a company cannot effectively restrict the users of a free program by obtaining a restrictive license from a patent holder. Therefore, we insist that any patent license obtained for a version of the library must be consistent with the full freedom of use specified in this license.

Most GNU software, including some libraries, is covered by the ordinary GNU General Public License. This license, the GNU Lesser General Public License, applies to certain designated libraries, and is quite different from the ordinary General Public License. We use this license for certain libraries in order to permit linking those libraries into non-free programs.

When a program is linked with a library, whether statically or using a shared library, the combination of the two is legally speaking a combined work, a derivative of the original library. The ordinary General Public License therefore permits such linking only if the entire combination fits its criteria of freedom. The Lesser General Public License permits more lax criteria for linking other code with the library.

We call this license the "Lesser" General Public License because it does Less to protect the user's freedom than the ordinary General Public License. It also provides other free software developers Less of an advantage over competing non-free programs. These disadvantages are the reason we use the ordinary General Public License for many libraries. However, the Lesser license provides advantages in certain special circumstances.

For example, on rare occasions, there may be a special need to encourage the widest possible use of a certain library, so that it becomes a de-facto standard. To achieve this, non-free programs must be allowed to use the library. A more frequent case is that a free library does the same job as widely used non-free libraries. In this case, there is little to gain by limiting the free library to free software only, so we use the Lesser General Public License.

In other cases, permission to use a particular library in non-free programs enables a greater number of people to use a large body of free software. For example, permission to use the GNU C Library in non-free programs enables many more people to use the whole GNU operating system, as well as its variant, the GNU/Linux operating system.

Although the Lesser General Public License is Less protective of the users' freedom, it does ensure that the user of a program that is linked with the Library has the freedom and the wherewithal to run that program using a modified version of the Library.

The precise terms and conditions for copying, distribution and modification follow. Pay close attention to the difference between a "work based on the library" and a "work that uses the library". The former contains code derived from the library, whereas the latter must be combined with the library in order to run.

2.0.3 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

0. This License Agreement applies to any software library or other program which contains a notice placed by the copyright holder or other authorized party saying it may be distributed under the terms of this Lesser General Public License (also called "this License"). Each licensee is addressed as "you".

A "library" means a collection of software functions and/or data prepared so as to be conveniently linked with application programs (which use some of those functions and data) to form executables.

The "Library", below, refers to any such software library or work which has been distributed under these terms. A "work based on the Library" means either the Library or any derivative work under copyright law: that is to say, a work containing the Library or a portion of it, either verbatim or with modifications and/or translated straightforwardly into another language. (Hereinafter, translation is included without limitation in the term "modification".)

"Source code" for a work means the preferred form of the work for making modifications to it. For a library, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the library.

Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running a program using the Library is not restricted, and output from such a program is covered only if its contents constitute a work based on the Library (independent of the use of the Library in a tool for writing it). Whether that is true depends on what the Library does and what the program that uses the Library does.

1. You may copy and distribute verbatim copies of the Library's complete source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and distribute a copy of this License along with the Library.

You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

2. You may modify your copy or copies of the Library or any portion of it, thus forming a work based on the Library, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

- **a)** The modified work must itself be a software library.
- **b)** You must cause the files modified to carry prominent notices stating that you changed the files and the date of any change.
- **c)** You must cause the whole of the work to be licensed at no charge to all third parties under the terms of this License.
- **d)** If a facility in the modified Library refers to a function or a table of data to be supplied by an application program that uses the facility, other than as an argument passed when the facility is invoked, then you must make a good faith effort to ensure that, in the event an application does not supply such function or table, the facility still operates, and performs whatever part of its purpose remains meaningful.

  (For example, a function in a library to compute square roots has a purpose that is entirely well-defined independent of the application. Therefore, Subsection 2d requires that any application-supplied function or table used by this function must be optional: if the application does not supply it, the square root function must still compute square roots.)

These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Library, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Library, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Library.

In addition, mere aggregation of another work not based on the Library with the Library (or with a work based on the Library) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

3. You may opt to apply the terms of the ordinary GNU General Public License instead of this License to a given copy of the Library. To do this, you must alter all the notices that refer to this License, so that they refer to the ordinary GNU General Public License, version 2, instead of to this License. (If a newer version than version 2 of the ordinary GNU General Public License has appeared, then you can specify that version instead if you wish.) Do not make any other change in these notices.

Once this change is made in a given copy, it is irreversible for that copy, so the ordinary GNU General Public License applies to all subsequent copies and derivative works made from that copy.

This option is useful when you wish to copy part of the code of the Library into a program that is not a library.

4. You may copy and distribute the Library (or a portion or derivative of it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange.

If distribution of object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place satisfies the requirement to distribute the source code, even though third parties are not compelled to copy the source along with the object code.

5. A program that contains no derivative of any portion of the Library, but is designed to work with the Library by being compiled or linked with it, is called a "work that uses the Library". Such a work, in isolation, is not a derivative work of the Library, and therefore falls outside the scope of this License.

However, linking a "work that uses the Library" with the Library creates an executable that is a derivative of the Library (because it contains portions of the Library), rather than a "work that uses the library". The executable is therefore covered by this License. Section 6 states terms for distribution of such executables.

When a "work that uses the Library" uses material from a header file that is part of the Library, the object code for the work may be a derivative work of the Library even though the source code is not. Whether this is true is especially significant if the work can be linked without the Library, or if the work is itself a library. The threshold for this to be true is not precisely defined by law.

If such an object file uses only numerical parameters, data structure layouts and accessors, and small macros and small inline functions (ten lines or less in length), then the use of the object file is unrestricted, regardless of whether it is legally a derivative work. (Executables containing this object code plus portions of the Library will still fall under Section 6.)

Otherwise, if the work is a derivative of the Library, you may distribute the object code for the work under the terms of Section 6. Any executables containing that work also fall under Section 6, whether or not they are linked directly with the Library itself.

6. As an exception to the Sections above, you may also combine or link a "work that uses the Library" with the Library to produce a work containing portions of the Library, and distribute that work under terms of your choice, provided that the terms permit modification of the work for the customer's own use and reverse engineering for debugging such modifications.

You must give prominent notice with each copy of the work that the Library is used in it and that the Library and its use are covered by this License. You must supply a copy of this License. If the work during execution displays copyright notices, you must include the copyright notice for the Library among them, as well as a reference directing the user to the copy of this License. Also, you must do one of these things:

- **a)** Accompany the work with the complete corresponding machine-readable source code for the Library including whatever changes were used in the work (which must be distributed under Sections 1 and 2 above); and, if the work is an executable linked with the Library, with the complete machine-readable "work that uses the Library", as object code and/or source code, so that the user can modify the Library and then relink to produce a modified executable containing the modified Library. (It is understood that the user who changes the contents of definitions files in the Library will not necessarily be able to recompile the application to use the modified definitions.)
- **b)** Use a suitable shared library mechanism for linking with the Library. A suitable mechanism is one that (1) uses at run time a copy of the library already present on the user's computer system, rather than copying library functions into the executable, and (2) will operate properly with a modified version of the library, if the user installs one, as long as the modified version is interface-compatible with the version that the work was made with.
- **c)** Accompany the work with a written offer, valid for at least three years, to give the same user the materials specified in Subsection 6a, above, for a charge no more than the cost of performing this distribution.
- **d)** If distribution of the work is made by offering access to copy from a designated place, offer equivalent access to copy the above specified materials from the same place.
- **e)** Verify that the user has already received a copy of these materials or that you have already sent this user a copy.

For an executable, the required form of the "work that uses the Library" must include any data and utility programs needed for reproducing the executable from it. However, as a special exception, the materials to be distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

It may happen that this requirement contradicts the license restrictions of other proprietary libraries that do not normally accompany the operating system. Such a contradiction means you cannot use both them and the Library together in an executable that you distribute.

7. You may place library facilities that are a work based on the Library side-by-side in a single library together with other library facilities not covered by this License, and distribute such a combined library, provided that the separate distribution of the work based on the Library and of the other library facilities is otherwise permitted, and provided that you do these two things:

- **a)** Accompany the combined library with a copy of the same work based on the Library, uncombined with any other library facilities. This must be distributed under the terms of the Sections above.
- **b)** Give prominent notice with the combined library of the fact that part of it is a work based on the Library, and explaining where to find the accompanying uncombined form of the same work.

8. You may not copy, modify, sublicense, link with, or distribute the Library except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense, link with, or distribute the Library is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

9. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Library or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Library (or any work based on the Library), you indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Library or works based on it.

10. Each time you redistribute the Library (or any work based on the Library), the recipient automatically receives a license from the original licensor to copy, distribute, link with or modify the Library subject to these terms and

conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties with this License.

11. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Library at all. For example, if a patent license would not permit royalty-free redistribution of the Library by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Library.

If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply, and the section as a whole is intended to apply in other circumstances.

It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.

12. If the distribution and/or use of the Library is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Library under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.

13. The Free Software Foundation may publish revised and/or new versions of the Lesser General Public License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns.

Each version is given a distinguishing version number. If the Library specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Library does not specify a license version number, you may choose any version ever published by the Free Software Foundation.

14. If you wish to incorporate parts of the Library into other free programs whose distribution conditions are incompatible with these, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

15. BECAUSE THE LIBRARY IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE LIBRARY, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE LIBRARY "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE LIBRARY IS WITH YOU. SHOULD THE LIBRARY PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.

16. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE LIBRARY AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE LIBRARY (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE LIBRARY TO OPERATE WITH ANY OTHER SOFTWARE), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

2.0.4 END OF TERMS AND CONDITIONS

2.0.5 How to Apply These Terms to Your New Libraries

If you develop a new library, and you want it to be of the greatest possible use to the public, we recommend making it free software that everyone can redistribute and change. You can do so by permitting redistribution under these terms (or, alternatively, under the terms of the ordinary General Public License).

To apply these terms, attach the following notices to the library. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the library's name and an idea of what it does.
Copyright (C) year  name of author

This library is free software; you can redistribute it and/or
modify it under the terms of the GNU Lesser General Public
License as published by the Free Software Foundation; either
version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public
License along with this library; if not, write to the Free Software
Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA  02110-1301  USA
```

Also add information on how to contact you by electronic and paper mail.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the library, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright interest in
the library 'Frob' (a library for tweaking knobs) written
by James Random Hacker.

signature of Ty Coon, 1 April 1990
Ty Coon, President of Vice
```

That's all there is to it!

Chapter 3

Building the documentation

The documentation for CASC can be generated locally using `Doxygen`. You must have a working copy of doxygen installed on your machine in order to build the documentation.

If CMake is able to find your doxygen installation then the following sequence of commands will build the basic documentation.

```
cmake ..
make casc_doc
```

3.0.1 Documentation for Developers

If you are contributing to or modifying the CASC library you may wish to document private class members or currently hidden metatemplate helper functions. Whether or not documentation for these items is generated can be controlled by modifying the default doxygen configuration: `doc/Doxyfile.in`.

To document private class functions and members toggle: `EXTRACT_PRIVATE = YES`

To enable metatemplate helper functions enable the conditional: `ENABLED_SECTIONS = detail`

Chapter 4

Frequently Asked Questions

1. Why is my simplex data not storing correctly?

If you are retrieving the data from the `SimplexID` using the dereference operator, you must retrieve the result as a reference in order to modify it. See the following example.

```
MeshType mesh = MeshType();
int key = mesh.insert({1}, 10);
auto data = *mesh.get_simplex_up({key});
data = 5;
std::cout «*mesh.get_simplex_up({key}); « std::endl // Prints 10

auto &dataRef = *mesh.get_simplex_up({key});
dataRef = 5;
std::cout « *mesh.get_simplex_up({key}) « std::endl // Prints 5
```

Chapter 5

Namespace Index

5.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

casc
Namespace for everything CASC 23
index_tracker
Index tracker namespace 42
index_tracker::index_tracker_detail
B-tree internal data structures 43
util
Metatemplate programming utilities namespace 44

Chapter 6

Data Structure Index

6.1 Data Structures

Here are the data structures with brief descriptions:

index_tracker::index_tracker_detail::BTreeNode< _T, _d >
An array based BTree . . . 49
casc::simplicial_complex< traits >::EdgeID< k >
External reference to an edge or a connection within the complex . . . 50
index_tracker::index_tracker< _T, _d >
Tracker of available indices implemented as a B-tree of intervals . . . 53
util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >
Maps an integer sequence and typename, F, into outholder . . . 54
index_tracker::index_tracker_detail::Interval< T >
Interval object represents a range . . . 55
casc::Orientable
Class representing the orientation . . . 56
util::range< T >
A range object to support range based for loops . . . 56
util::remove_first_val< Integer, IntegerSequence >
General template for removing the first value from a type holder . . . 58
util::remove_first_val< Integer, InHolder< Integer, I, Is... > >
Specialization for removing first integer from a sequence of compile time integers . . . 59
util::reverse_sequence< Integer, IntegerSequence >
Reverse an Integer Sequence . . . 59
casc::simplicial_complex< traits >::SimplexID< k >
A handle for a simplex object in the complex . . . 60
casc::SimplexMap< Complex >
A multimap to represent a map of simplex indices to a set of simplices . . . 65
casc::SimplexSet< Complex >
A multiset to store simplices in a simplicial_complex . . . 68
casc::simplicial_complex< traits >
The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring . . . 74
util::type_get< k, T >
Helper to get the kth element from a type_holder . . . 98
util::type_get< 0, type_holder< Ts... > >
Specialization for terminal case . . . 99
util::type_get< k, type_holder< Ts... > >
Specialization to recursively pop types to get the kth type . . . 99

util::type_holder< Ts >
Queue based data structure to hold list of types 100
util::type_holder< T, Ts... >
Partial specialization to allow FIFO access of typenames 100
util::type_map< C, V >
Map the types in C into V<T> 101
util::type_swap< TUPLE, HOLDER_FULL >
Move a list of types from one container to another 102
util::type_swap< TUPLE, HOLDER< Ts... > >
Move a list of types from one container to another 102

Chapter 7

File Index

7.1 File List

Here is a list of all documented files with brief descriptions:

include/casc/CASCFunctions.h
: Contains various functions that operate on simplicial complexes . . . 105

include/casc/CASCTraversals.h
: Implementations of various advanced traversals such as by neighborhood and breadth first search . . . 111

include/casc/decimate.h
: Meta-data aware decimation functions . . . 122

include/casc/index_tracker.h
: B-tree based interval tracker . . . 130

include/casc/Orientable.h
: Data type for orientability . . . 143

include/casc/SimplexMap.h
: SimplexMap data structure and associated convenience functions . . . 147

include/casc/SimplexSet.h
: SimplexSet data structure and associated convenience functions . . . 150

include/casc/SimplicialComplex.h
: This header contains the main CASC data structure and associated components . . . 158

include/casc/stringutil.h
: String utilities for CASC . . . 192

include/casc/typetraits.h
: Helper functions for debugging template types . . . 193

include/casc/util.h
: Metatemplate pack expansion helpers . . . 195

Chapter 8

Namespace Documentation

8.1 casc Namespace Reference

Namespace for everything CASC.

Data Structures

- struct Orientable

  Class representing the orientation.
- struct SimplexMap

  A multimap to represent a map of simplex indices to a set of simplices.
- struct SimplexSet

  A multiset to store simplices in a simplicial_complex.
- class simplicial_complex

  The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring.

Typedefs

- template<typename KeyType , typename ... Ts>
  using AbstractSimplicialComplex = simplicial_complex< detail::simplicial_complex_traits_default< KeyType, Ts... > >
- template<typename T >
  using **NodeSet** = std::unordered_set< T, simplex_set_detail::hashSimplexID< T > >

  Helpful alias defining a unordered_set of simplices. See also hashSimplexID.

Functions

- template<typename Complex >
  void getStar (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the star of a SimplexSet.
- template<typename Complex , typename Simplex >
  void getStar (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Gets the star of a simplex.

- template<typename Complex >
  void getClosure (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the closure of a simplex set.
- template<typename Complex , typename Simplex >
  void getClosure (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Compute the closure of a simplex.
- template<typename Complex >
  void getLink (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the link of a SimplexSet.
- template<typename Complex , typename Simplex >
  void getLink (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Gets the link of a simplex.
- template<typename Complex >
  void writeDOT (const std::string &filename, Complex &F)

  Writes out the topology of an ASC into the dot format.
- template<typename Visitor , typename SimplexID >
  void visit_BFS_up (Visitor &&v, typename SimplexID::complex &F, SimplexID s)

  Traverse BFS up the complex and apply a visitor function to each simplex visited.
- template<typename Visitor , typename SimplexID >
  void visit_BFS_down (Visitor &&v, typename SimplexID::complex &F, SimplexID s)

  Traverse BFS down the complex and apply a visitor function to each simplex visited.
- template<typename Visitor , typename EdgeID >
  void edge_up (Visitor &&v, typename EdgeID::complex &F, EdgeID s)

  Traverse across edges BFS.
- template<class Complex , std::size_t level, class InsertIter >
  void neighbors (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)

  Push the immediate face neighbors into the provided iterator.
- template<class Complex , class SimplexID , class InsertIter >
  void neighbors (Complex &F, SimplexID nid, InsertIter iter)

  This is a helper function to assist neighbors to automatically deduce the integral level.
- template<class Complex , std::size_t level, class InsertIter >
  void neighbors_up (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)

  Push the immediate coface neighbors into the provided iterator.
- template<class Complex , class SimplexID , class InsertIter >
  void neighbors_up (Complex &F, SimplexID nid, InsertIter iter)

  This is a helper function to assist neighbors to automatically deduce the integral level.
- template<class Complex , std::size_t level, typename Iterator >
  void kneighbors_up (Complex &F, int ring, std::set< typename Complex::template SimplexID< level > > &nbors, Iterator begin, Iterator end)

  Code for returning a set of k-ring neighbors.
- template<class Complex , class SimplexID >
  void kneighbors_up (Complex &F, SimplexID nid, int ring, std::set< SimplexID > &nbors)

  Helper function to help kneighbors_up to deduce the integral level of SimplexID.
- template<class Complex , std::size_t level, typename Iterator >
  void kneighbors (Complex &F, int ring, std::set< typename Complex::template SimplexID< level > > &nbors, Iterator begin, Iterator end)

  Code for returning a set of k-ring neighbors.
- template<class Complex , class SimplexID >
  void kneighbors (Complex &F, SimplexID nid, int ring, std::set< SimplexID > &nbors)

  Helper function to help kneighbors to deduce the integral level of SimplexID.
- template<typename Complex >
  void perform_removal (Complex &F, casc::SimplexSet< Complex > &S)

  Remove simplex in SimplexSet S from complex F.

- template<typename Complex >
  void perform_insertion (Complex &F, typename decimation_detail::SimplexDataSet< Complex >::type &S)

  Insert all simplices in SimplexSet S into complex F
- template<typename Complex , template< typename > class Callback>
  void run_user_callback (Complex &F, casc::SimplexMap< Complex > &S, Callback< Complex > &&clbk, typename decimation_detail::SimplexDataSet< Complex >::type &rv)

  Run the user specified callback function.
- template<typename Complex , typename Simplex , template< typename > class Callback>
  void decimate (Complex &F, Simplex s, Callback< Complex > &&clbk)

  Decimate a simplex of any dimension while considering any meta-data stores on decimated simplices.
- template<typename Complex , typename Simplex >
  Complex::KeyType decimateFirstHalf (Complex &F, Simplex s, SimplexMap< Complex > &simplexMap)

  Given a simplex to decimate generate a pre-post mapping.
- template<typename Complex >
  void decimateBackHalf (Complex &F, SimplexMap< Complex > &simplexMap, typename decimation_↩ detail::SimplexDataSet< Complex >::type &rv)

  Given a simplexMap and mapped resulting data execute the decimation.
- template<typename Complex >
  void init_orientation (Complex &F)

  Initialize the partial ordering of the simplex edges.
- template<typename Complex >
  void clear_orientation (Complex &F)

  Clear the orientation of the facets.
- template<typename Complex >
  std::tuple< int, bool, bool > compute_orientation (Complex &F)

  Initializes and calculates the orientation of a simplicial_complex.
- template<typename Complex >
  std::tuple< int, bool, bool > check_orientation (Complex &F)

  Checks for self consistent orientation and fill in missing orientations.
- template<std::size_t k, typename Complex >
  static auto & get (SimplexMap< Complex > &S)

  Get the map for a simplex dimension.
- template<std::size_t k, typename Complex >
  static auto & **get** (const SimplexMap< Complex > &S)

  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- template<std::size_t k, typename Complex >
  static auto & get (SimplexSet< Complex > &S)

  Get the NodeSet for a simplex dimension from a SimplexSet.
- template<std::size_t k, typename Complex >
  static auto & get (const SimplexSet< Complex > &S)
- template<typename Complex >
  bool operator== (const SimplexSet< Complex > &lhs, const SimplexSet< Complex > &rhs)

  Compare if the sets are equivalent.
- template<typename Complex >
  bool operator!= (const SimplexSet< Complex > &lhs, const SimplexSet< Complex > &rhs)

  Compare if the sets are not equivalent.
- template<typename Complex >
  static void set_union (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

  Compute the set union.
- template<typename Complex >
  static void set_intersection (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

Compute the set intersection.

- template<typename Complex >
  static void set_difference (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

  Compute the set difference.
- template<typename T , std::size_t k>
  std::string to_string (const std::array< T, k > &A)

  Returns a string representation of the vertex subsimplicies of a given simplex.

8.1.1 Typedef Documentation

8.1.1.1 AbstractSimplicialComplex

```
template<typename KeyType , typename ...  Ts>
using casc::AbstractSimplicialComplex = typedef simplicial_complex< detail::simplicial_↩
complex_traits_default<KeyType, Ts...> >
```

Alias to generate a CASC from a list of traits. See also simplicial_complex_traits_default. Example – To create a tetrahedral mesh with integer data on all simplices:

```
auto mesh = AbstractSimplicialComplex<
    int, // KEYTYPE
    int, // Root data
    int, // Vertex data
    int, // Edge data
    int, // Face data
    int  // Volume data
>();
```

8.1.2 Function Documentation

8.1.2.1 check_orientation()

```
template<typename Complex >
std::tuple< int, bool, bool > casc::check_orientation (
            Complex & F )
```

Parameters

F	Simplicial_complex

Template Parameters

Complex	Typename of the simplicial_complex.

Returns

A tuple of the number of connected components, where the complex is orientable, and if it is psuedo manifold.

8.1.2.2 clear_orientation()

```
template<typename Complex >
void casc::clear_orientation (
            Complex & F )
```

Parameters

F	Simplicial complex of interest

Template Parameters

Complex	Typename of the simplicial complex

8.1.2.3 compute_orientation()

```
template<typename Complex >
std::tuple< int, bool, bool > casc::compute_orientation (
            Complex & F )
```

Parameters

F	Simplicial_complex

Template Parameters

Complex	Typename of the simplicial_complex.

Returns

A tuple of the number of connected components, where the complex is orientable, and if it is psuedo manifold.

8.1.2.4 decimate()

```
template<typename Complex , typename Simplex , template< typename > class Callback>
void casc::decimate (
            Complex & F,
            Simplex s,
            Callback< Complex > && clbk )
```

Parameters

in	*F*	simplicial_complex to operate on.
in	*s*	Simplex to decimate.
in	*clbk*	Callback function to map meta-data

Template Parameters

Complex	Typename of the simplicial_complex
Simplex	Typename of the simplex
Callback	Typename of the template template callback functor

Alias for SimplexSet

Alias for SimplexMap

8.1.2.5 decimateBackHalf()

```
template<typename Complex >
void casc::decimateBackHalf (
            Complex & F,
            SimplexMap< Complex > & simplexMap,
            typename decimation_detail::SimplexDataSet< Complex >::type & rv )
```

Parameters

F	Simplicial complex to operate on
simplexMap	SimplexMap mapping simplices before and after decimation
rv	Resulting data for each simplex

Template Parameters

Complex	Typename of the complex of interest

8.1.2.6 decimateFirstHalf()

```
template<typename Complex , typename Simplex >
Complex::KeyType casc::decimateFirstHalf (
            Complex & F,
            Simplex s,
            SimplexMap< Complex > & simplexMap )
```

Parameters

in	*F*	simplicial_complex to operate on.
in	*s*	Simplex to decimate.
	simplexMap	The simplex map to populate

Template Parameters

Complex	Typename of the simplicial_complex
Simplex	Typename of the simplex

Alias for SimplexSet

8.1.2.7 edge_up()

```
template<typename Visitor , typename EdgeID >
void casc::edge_up (
            Visitor && v,
            typename EdgeID::complex & F,
            EdgeID s )
```

Parameters

in	*v*	Visitor functor to apply.
	F	The simplicial_complex to traverse.
in	*s*	The edge to start at.

Template Parameters

Visitor	Typename of the functor.
EdgeID	Typename of the edge.

8.1.2.8 get() [1/3]

```
template<std::size_t k, typename Complex >
static auto & casc::get (
            const SimplexSet< Complex > & S )  [inline], [static]
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

8.1.2.9 get() [2/3]

```
template<std::size_t k, typename Complex >
static auto & casc::get (
            SimplexMap< Complex > & S )  [inline], [static]
```

Parameters

S	SimplexMap to retrieve from.

Template Parameters

k	Simplex dimension.
Complex	Typename of the complex.

Returns

Returns a map of std::Array<KeyType, k> to SimplexSet.

8.1.2.10 get() [3/3]

```
template<std::size_t k, typename Complex >
static auto & casc::get (
            SimplexSet< Complex > & S )  [inline], [static]
```

Parameters

S	SimplexSet of interest.

Template Parameters

k	Simplex dimension desired.
Complex	Typename of the simplicial_complex.

Returns

A NodeSet which holds simplices of dimension 'k' and a member of SimplexSet 'S'.

8.1.2.11 getClosure() [1/2]

```
template<typename Complex >
void casc::getClosure (
            Complex & F,
            casc::SimplexSet< Complex > & S,
            casc::SimplexSet< Complex > & dest )
```

Parameters

in	*F*	Complex of interest.
in	*S*	SimplexSet to compute the closure of.
out	*dest*	Destination SimplexSet

Template Parameters

Complex	Typename of the complex.

8.1.2.12 getClosure() [2/2]

```
template<typename Complex , typename Simplex >
void casc::getClosure (
            Complex & F,
            Simplex & s,
            casc::SimplexSet< Complex > & dest )
```

Parameters

in	*F*	Complex of interest.
in	*s*	Simplex of interest.
out	*dest*	Destination SimplexSet.

Template Parameters

Complex	Typename of the complex.
Simplex	Typename of the simplex.

8.1.2.13 getLink() [1/2]

```
template<typename Complex >
void casc::getLink (
            Complex & F,
            casc::SimplexSet< Complex > & S,
            casc::SimplexSet< Complex > & dest )
```

Parameters

in	*F*	Complex of interest.
in	*S*	SimplexSet to get the link of.
out	*dest*	Destination SimplexSet.

Template Parameters

Complex	Typename of the complex.

8.1.2.14 getLink() [2/2]

```
template<typename Complex , typename Simplex >
void casc::getLink (
            Complex & F,
            Simplex & s,
            casc::SimplexSet< Complex > & dest )
```

Parameters

F	Complex of interest.
s	Simplex of interest.
dest	Destination SimplexSet.

Template Parameters

Complex	Typename of the complex.
Simplex	Typename of the simplex.

8.1.2.15 getStar() [1/2]

```
template<typename Complex >
void casc::getStar (
            Complex & F,
            casc::SimplexSet< Complex > & S,
            casc::SimplexSet< Complex > & dest )
```

Parameters

in	*F*	Complex of interest.
in	*S*	SimplexSet to compute the star of.
out	*dest*	Destination SimplexSet.

Template Parameters

Complex	Typename of the complex.

8.1.2.16 getStar() [2/2]

```
template<typename Complex , typename Simplex >
void casc::getStar (
            Complex & F,
            Simplex & s,
            casc::SimplexSet< Complex > & dest )
```

Parameters

in	*F*	Complex of interest.
	s	Simplex to get the star of.
out	*dest*	Destination SimplexSet.

Template Parameters

Complex	Typename of the complex.
Simplex	Typename of the simplex.

8.1.2.17 init_orientation()

```
template<typename Complex >
void casc::init_orientation (
            Complex & F )
```

Parameters

F	Simplicial complex of interest

Template Parameters

Complex	Typename of the simplicial complex

8.1.2.18 kneighbors() [1/2]

```
template<class Complex , std::size_t level, typename Iterator >
void casc::kneighbors (
            Complex & F,
            int ring,
            std::set< typename Complex::template SimplexID< level > > & nbors,
            Iterator begin,
            Iterator end )
```

Parameters

in	*F*	The simplicial_complex to traverse.
in	*ring*	The number of rings of neighbors to collect.
out	*nbors*	Set of previously seen simplices.
in	*begin*	The begin
in	*end*	The end

Template Parameters

Complex	Typename of the simplicial_complex.
level	Simplex dimension of the simplex and neighbors.
Iterator	{ description }

8.1.2.19 kneighbors() [2/2]

```
template<class Complex , class SimplexID >
void casc::kneighbors (
            Complex & F,
            SimplexID nid,
            int ring,
            std::set< SimplexID > & nbors )
```

Parameters

in	*F*	The simplicial complex
in	*nid*	Simplex of interest to get the nieghbors of.
in	*ring*	The number of rings to include as a neighbor.
out	*nbors*	Set of neighbors to populate.

Template Parameters

Complex	Typename of the complex.
SimplexID	Typename of the SimplexID.

8.1.2.20 kneighbors_up() [1/2]

```
template<class Complex , std::size_t level, typename Iterator >
void casc::kneighbors_up (
            Complex & F,
            int ring,
            std::set< typename Complex::template SimplexID< level > > & nbors,
            Iterator begin,
            Iterator end )
```

Parameters

in	*F*	The simplicial_complex to traverse.
in	*ring*	The number of rings of neighbors to collect.
out	*nbors*	Set of previously seen simplices.
in	*begin*	The begin
in	*end*	The end

Template Parameters

Complex	Typename of the simplicial_complex.
level	Simplex dimension of the simplex and neighbors.
Iterator	{ description }

8.1.2.21 kneighbors_up() [2/2]

```
template<class Complex , class SimplexID >
void casc::kneighbors_up (
            Complex & F,
            SimplexID nid,
            int ring,
            std::set< SimplexID > & nbors )
```

Parameters

in	*F*	The simplicial complex
in	*nid*	Simplex of interest to get the nieghbors of.
in	*ring*	The number of rings to include as a neighbor.
out	*nbors*	Set of neighbors to populate.

Template Parameters

Complex	Typename of the complex.
SimplexID	Typename of the SimplexID.

8.1.2.22 neighbors() [1/2]

```
template<class Complex , class SimplexID , class InsertIter >
void casc::neighbors (
            Complex & F,
            SimplexID nid,
            InsertIter iter )
```

Parameters

	F	The simplicial complex.
in	*nid*	Simplex to get neighbors of.
in	*iter*	The iterator to push members into.

Template Parameters

Complex	Type of the simplicial complex

Template Parameters

level	The integral level of the node
InsertIter	Typename of the iterator.

8.1.2.23 neighbors() [2/2]

```
template<class Complex , std::size_t level, class InsertIter >
void casc::neighbors (
            Complex & F,
            typename Complex::template SimplexID< level > nid,
            InsertIter iter )
```

This function gets the set of neighbors which share a common face. We compute this by traversing to all faces of the simplex of interest. Then we get all cofaces of this set. Depending on the type of iterator passed, duplicate simplices will be included or excluded. Note that this is the traditional definition of neighbor. For example, faces which share an edge are neighbors.

Parameters

	F	The simplicial complex
in	*nid*	Simplex to get neighbors of.
in	*iter*	The iterator to push members into.

Template Parameters

Complex	Type of the simplicial complex
level	The integral level of the node
InsertIter	Typename of the iterator.

8.1.2.24 neighbors_up() [1/2]

```
template<class Complex , class SimplexID , class InsertIter >
void casc::neighbors_up (
            Complex & F,
            SimplexID nid,
            InsertIter iter )
```

Parameters

	F	The simplicial complex.
in	*nid*	Simplex to get neighbors of.
in	*iter*	The iterator to push members into.

Template Parameters

Complex	Type of the simplicial complex
level	The integral level of the node
InsertIter	Typename of the iterator.

8.1.2.25 neighbors_up() [2/2]

```
template<class Complex , std::size_t level, class InsertIter >
void casc::neighbors_up (
            Complex & F,
            typename Complex::template SimplexID< level > nid,
            InsertIter iter )
```

Parameters

	F	The simplicial complex.
in	*nid*	Simplex to get neighbors of.
in	*iter*	The iterator to push members into.

Template Parameters

Complex	Type of the simplicial complex
level	The integral level of the node
InsertIter	Typename of the iterator.

8.1.2.26 operator"!=()

```
template<typename Complex >
bool casc::operator!= (
            const SimplexSet< Complex > & lhs,
            const SimplexSet< Complex > & rhs )
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

Complex	Typename of the simplicial_complex.

Returns

True if the sets are inequal, false otherwise.

8.1.2.27 operator==()

```
template<typename Complex >
bool casc::operator== (
            const SimplexSet< Complex > & lhs,
            const SimplexSet< Complex > & rhs )
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

Complex	Typename of the simplicial_complex

Returns

True if the sets are equal, false otherwise.

8.1.2.28 perform_insertion()

```
template<typename Complex >
void casc::perform_insertion (
            Complex & F,
            typename decimation_detail::SimplexDataSet< Complex >::type & S )
```

Parameters

F	The simplicial_complex to insert into.
S	SimplexSet of simplices to insert.

Template Parameters

Complex	Typename of complex

8.1.2.29 perform_removal()

```
template<typename Complex >
void casc::perform_removal (
            Complex & F,
            casc::SimplexSet< Complex > & S )
```

Parameters

F	The simplicial_complex to remove from.
S	SimplexSet of simplices to remove.

Template Parameters

Complex	Typename of complex

8.1.2.30 run_user_callback()

```
template<typename Complex , template< typename > class Callback>
void casc::run_user_callback (
            Complex & F,
            casc::SimplexMap< Complex > & S,
            Callback< Complex > && clbk,
            typename decimation_detail::SimplexDataSet< Complex >::type & rv )
```

Parameters

in	*F*	The simplicial_complex
in	*S*	SimplexMap of
in	*clbk*	User specified callback functor
out	*rv*	Multi-vector to place results.

Template Parameters

Complex	Typename of the simplicial_complex
Callback	Typename of the template template callback functor

8.1.2.31 set_difference()

```
template<typename Complex >
static void casc::set_difference (
            const SimplexSet< Complex > & A,
            const SimplexSet< Complex > & B,
            SimplexSet< Complex > & dest )  [static]
```

Parameters

in	*A*	A SimplexSet
in	*B*	Another SimplexSet
out	*dest*	The destination SimplexSet.

Template Parameters

Complex	Typename of the simplicial_complex.

8.1.2.32 set_intersection()

```
template<typename Complex >
static void casc::set_intersection (
            const SimplexSet< Complex > & A,
            const SimplexSet< Complex > & B,
            SimplexSet< Complex > & dest )  [static]
```

Parameters

in	*A*	A SimplexSet
in	*B*	Another SimplexSet
out	*dest*	The destination SimplexSet.

Template Parameters

Complex	Typename of the simplicial_complex.

8.1.2.33 set_union()

```
template<typename Complex >
static void casc::set_union (
            const SimplexSet< Complex > & A,
            const SimplexSet< Complex > & B,
            SimplexSet< Complex > & dest )  [static]
```

Parameters

in	*A*	A SimplexSet
in	*B*	Another SimplexSet
out	*dest*	The destination SimplexSet.

Template Parameters

Complex	Typename of the simplicial_complex.

8.1.2.34 to_string()

```
template<typename T , std::size_t k>
std::string casc::to_string (
             const std::array< T, k > & A )
```

Parameters

in	*A*	Array containing name of a simplex.

Template Parameters

T	Typename KeyType.
k	Dimension of the simplex.

Returns

String representation of the object.

8.1.2.35 visit_BFS_down()

```
template<typename Visitor , typename SimplexID >
void casc::visit_BFS_down (
             Visitor && v,
             typename SimplexID::complex & F,
             SimplexID s )
```

Parameters

in	*v*	Visitor functor to apply.
	F	The simplicial_complex to traverse.
in	*s*	The simplex to start at.

Template Parameters

Visitor	Typename of the functor.
SimplexID	Typename of the simplex.

8.1.2.36 visit_BFS_up()

```
template<typename Visitor , typename SimplexID >
void casc::visit_BFS_up (
            Visitor && v,
            typename SimplexID::complex & F,
            SimplexID s )
```

Parameters

in	*v*	Visitor functor to apply.
	F	The simplicial_complex to traverse.
in	*s*	The simplex to start at.

Template Parameters

Visitor	Typename of the functor.
SimplexID	Typename of the simplex.

8.1.2.37 writeDOT()

```
template<typename Complex >
void casc::writeDOT (
            const std::string & filename,
            Complex & F )
```

The resulting dot file can be rendered into an image using tools such as GraphViz.
`dot -Tpng input.dot > output.png`

Parameters

in	*filename*	Filename to write out to.
in	*F*	Simplicial complex to generate the DOT of.

Template Parameters

Complex	Typename of the simplicial complex.

8.2 index_tracker Namespace Reference

Index tracker namespace.

Namespaces

- namespace index_tracker_detail

  B-tree internal data structures.

Data Structures

- class index_tracker

  Tracker of available indices implemented as a B-tree of intervals.

Functions

- template<typename T , std::size_t d>
  std::ostream & **operator**<< (std::ostream &out, const index_tracker_detail::BTreeNode< T, d > *head)

8.3 index_tracker::index_tracker_detail Namespace Reference

B-tree internal data structures.

Data Structures

- struct BTreeNode

  An array based BTree.
- struct Interval

  Interval object represents a range.

Typedefs

- template<typename Node >
  using **Pointer** = typename Node::Pointer
- template<typename Node >
  using **Data** = typename Node::Data
- template<typename Node >
  using **Scalar** = typename Node::Scalar

Functions

- template<typename T >
  bool **operator**< (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  bool **operator**> (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  bool **operator**< (T x, const Interval< T > &y)
- template<typename T >
  bool **operator**> (const Interval< T > &x, T y)
- template<typename T >
  bool **operator**< (const Interval< T > &x, T y)
- template<typename T >
  bool **operator**> (T x, const Interval< T > &y)
- template<typename T >
  bool **operator==** (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  std::ostream & **operator**<< (std::ostream &out, const Interval< T > &x)

- template<typename T >
  int **merge** (Interval< T > &A, T x)
- template<typename Node >
  void **rebalance** (Pointer< Node > head, std::size_t i)
- template<typename Node >
  void **insert_H** (Pointer< Node > head, const Data< Node > &data)
- template<typename Node >
  Pointer< Node > **insert** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  bool **get** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  void **get_replacement** (Pointer< Node > head, Data< Node > &key)
- template<typename Node >
  void **remove_H** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  Pointer< Node > **remove** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  void **fill_left** (Pointer< Node > head, Data< Node > &x)
- template<typename Node >
  void **fill_right** (Pointer< Node > head, Data< Node > &x)
- template<typename Node >
  void **insert_scalar_H** (Pointer< Node > head, Scalar< Node > data)
- template<typename Node >
  Pointer< Node > **insert_scalar** (Pointer< Node > head, Scalar< Node > data)
- template<typename Node >
  void **insert_left** (Pointer< Node > head, const Data< Node > &x)
- template<typename Node >
  bool **remove_scalar_H** (Pointer< Node > head, Scalar< Node > x)
- template<typename Node >
  bool **remove_scalar** (Pointer< Node > &head, Scalar< Node > data)
- template<typename Node >
  Scalar< Node > **pop_scalar** (Pointer< Node > &head)
- template<typename Node >
  void **destruct** (Pointer< Node > head)
- template<typename Node >
  Data< Node > **check_order** (Pointer< Node > head, Data< Node > curr)

8.4 util Namespace Reference

Metatemplate programming utilities namespace.

Data Structures

- struct int_type_map

  Maps an integer sequence and typename, F, into outholder.
- struct range

  A range object to support range based for loops.
- struct remove_first_val

  General template for removing the first value from a type holder.
- struct remove_first_val< Integer, InHolder< Integer, I, Is... > >

  Specialization for removing first integer from a sequence of compile time integers.
- struct reverse_sequence

Reverse an Integer Sequence.

- struct type_get

  Helper to get the kth element from a type_holder.
- struct type_get< 0, type_holder< Ts... > >

  Specialization for terminal case.
- struct type_get< k, type_holder< Ts... > >

  Specialization to recursively pop types to get the kth type.
- struct type_holder

  Queue based data structure to hold list of types.
- struct type_holder< T, Ts... >

  Partial specialization to allow FIFO access of typenames.
- struct type_map

  Map the types in C into `V<T>`.
- struct type_swap

  Move a list of types from one container to another.
- struct type_swap< TUPLE, HOLDER< Ts... > >

  Move a list of types from one container to another.

Functions

- template<typename T >
  range< T > make_range (T b, T e)

  Make a range object.
- template<typename T >
  range< T > make_range (std::pair< T, T > p)

  Makes a range object.
- template<class Integer , typename IntegerSequence , typename Fn , typename ... Args>
  void int_for_each (Fn &&f, Args &&... args)

  Calls a function `f.apply<k>()` for a sequence of integer k's.

8.4.1 Function Documentation

8.4.1.1 int_for_each()

```
template<class Integer , typename IntegerSequence , typename Fn , typename ...  Args>
void util::int_for_each (
            Fn && f,
            Args &&...  args )
```

Parameters

in	*args*	Arguments to f
in	*f*	Functor with `apply<k>()` method

Template Parameters

Integer	Integer type
IntegerSequence	Sequence of integers to iterate
Fn	Typename of functor f
Args	Typenames of the arguments

8.4.1.2 make_range() [1/2]

```
template<typename T >
range< T > util::make_range (
            std::pair< T, T > p )
```

Parameters

in	*p*	A pair containing begin and end iterators.

Template Parameters

T	Typename of the iterator.

Returns

Returns a range of the iterators.

8.4.1.3 make_range() [2/2]

```
template<typename T >
range< T > util::make_range (
            T b,
            T e )
```

Parameters

in	*b*	Iterator to the beginning.
in	*e*	Iterator to the end.

Template Parameters

T	Typename of the iterator.

Returns

Returns a range of the iterators.

Chapter 9

Data Structure Documentation

9.1 index_tracker::index_tracker_detail::BTreeNode< _T, _d > Struct Template Reference

An array based BTree.

`#include <index_tracker.h>`

Public Types

- using **Scalar** = _T
- using **Data** = Interval< Scalar >
- using **Pointer** = BTreeNode *

Public Member Functions

- **BTreeNode** (const Data &t)
- template<typename Iter >
  BTreeNode (Iter begin, Iter end)

Data Fields

- std::size_t **k**
- std::array< Data, N > **data**
- std::array< Pointer, N+1 > **next**

Static Public Attributes

- static constexpr std::size_t **d** = _d
- static constexpr std::size_t **N** = 2*d+1

9.1.1 Detailed Description

template<typename _T, std::size_t _d>
struct index_tracker::index_tracker_detail::BTreeNode< _T, _d >

Template Parameters

↩ _↩ *T*	{ description }
↩ _↩ *d*	{ description }

The documentation for this struct was generated from the following file:

- include/casc/index_tracker.h

9.2 casc::simplicial_complex< traits >::EdgeID< k > Struct Template Reference

External reference to an edge or a connection within the complex.

`#include <SimplicialComplex.h>`

Public Types

- using **complex** = simplicial_complex< traits >

  Typename of the complex.

Public Member Functions

- **EdgeID** ()

  Default contstructor wraps a nullptr and dummy edge.
- EdgeID (NodePtr< k > p, KeyType e)

  Constructor to wrap an Edge.
- EdgeID (const EdgeID &rhs)

  Copy constructor.
- EdgeID & **operator=** (const EdgeID &rhs)

  Assignment operator.
- auto const & **operator*** () const

  Dereferencing an EdgeID gets the data on the edge.
- auto & **operator*** ()

  Dereferencing an EdgeID gets the data on the edge.
- KeyType **key** () const

  Get the key of the edge.
- auto const & **data** () const

  Return the data stored on the edge.
- auto & **data** ()

  Return the data stored on the edge.
- SimplexID< k > up () const

  Get the coboundary simplex.
- SimplexID< k-1 > down () const

  Get the simplex below.

Data Fields

- friend **simplicial_complex< traits >**

  EdgeID is a friend of the complex.

Static Public Attributes

- static constexpr std::size_t **level** = k

  The dimension of the simplex which the edge points to.

Friends

- bool **operator==** (EdgeID lhs, EdgeID rhs)

  Equality of wrapped pointers and edges.
- bool **operator!=** (EdgeID lhs, EdgeID rhs)

  Compare wrapped pointers and edges.
- bool **operator<=** (EdgeID lhs, EdgeID rhs)

  Compare wrapped pointers and edges.
- bool **operator>=** (EdgeID lhs, EdgeID rhs)

  Compare wrapped pointers and edges.
- bool **operator<** (EdgeID lhs, EdgeID rhs)

  Less than defines an ordering of key types on the edges.
- bool **operator>** (EdgeID lhs, EdgeID rhs)

  Greater than comparison.

9.2.1 Detailed Description

template<typename traits>
template<std::size_t k>
struct casc::simplicial_complex< traits >::EdgeID< k >

Template Parameters

k	The edge connects a simplex of size k-1 to a simplex of size k.

9.2.2 Constructor & Destructor Documentation

9.2.2.1 EdgeID() [1/2]

```
template<typename traits >
template<std::size_t k>
casc::simplicial_complex< traits >::EdgeID< k >::EdgeID (
            NodePtr< k > p,
            KeyType e )  [inline]
```

Parameters

in	*p*	Pointer to the next Node.
in	*e*	Key of the edge

9.2.2.2 EdgeID() [2/2]

```
template<typename traits >
template<std::size_t k>
casc::simplicial_complex< traits >::EdgeID< k >::EdgeID (
            const EdgeID< k > & rhs )  [inline]
```

Parameters

in	*rhs*	The right hand side

9.2.3 Member Function Documentation

9.2.3.1 down()

```
template<typename traits >
template<std::size_t k>
SimplexID< k-1 > casc::simplicial_complex< traits >::EdgeID< k >::down ( ) const  [inline]
```

Returns

SimplexID of the simplex below the edge.

9.2.3.2 up()

```
template<typename traits >
template<std::size_t k>
SimplexID< k > casc::simplicial_complex< traits >::EdgeID< k >::up ( ) const  [inline]
```

Returns

SimplexID of the simplex above the edge.

The documentation for this struct was generated from the following file:

- include/casc/SimplicialComplex.h

9.3 index_tracker::index_tracker< _T, _d > Class Template Reference

Tracker of available indices implemented as a B-tree of intervals.

`#include <index_tracker.h>`

Public Types

- using **Node** = index_tracker_detail::BTreeNode< _T, _d >
  Typedef of BTree Node.
- using **T** = _T

Public Member Functions

- index_tracker ()
  Number of bins.
- void **insert** (T x)
- index_tracker_detail::Scalar< Node > **pop** ()
- void **remove** (index_tracker_detail::Scalar< Node > x)
- bool **empty** () const

Static Public Attributes

- static constexpr std::size_t **d** = _d
  Typename of the type to store.

Friends

- std::ostream & **operator**<< (std::ostream &out, const index_tracker &x)

9.3.1 Detailed Description

template<typename _T, std::size_t _d = 16>
class index_tracker::index_tracker< _T, _d >

Template Parameters

↩ _↩ *T*	Typename of the indices
↩ _↩ *d*	Max number of interval bins = 2∗value+1

9.3.2 Constructor & Destructor Documentation

9.3.2.1 index_tracker()

```
template<typename _T , std::size_t _d = 16>
index_tracker::index_tracker< _T, _d >::index_tracker ( )  [inline]
```

Initialize with interval [0∼max)

The documentation for this class was generated from the following file:

- include/casc/index_tracker.h

9.4 util::int_type_map< IntegerType, OutHolder, IntegerSequence, F > Struct Template Reference

Maps an integer sequence and typename, F, into outholder.

```
#include <util.h>
```

Public Types

- using **type** = typename detail::int_type_map_helper< IntegerType, OutHolder, IntegerSequence, F >::type
  Tuple of `Out<F<0>, F<1>, F<2>, ...>`.

9.4.1 Detailed Description

template<class IntegerType, template< class ... > class OutHolder, class IntegerSequence, template< IntegerType > class F> struct util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >

Given an Integer Sequence `I<0,1,2,3,...>` and template template type `F<I>`, this function produces `Out<F<0>, F<1>, F<2>, ...>`.

Template Parameters

IntegerType	Typename of an integer type
OutHolder	Typename of a holder for types
IntegerSequence	Integral sequence of types
F	Typename of class to be broadcast with integer

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.5 index_tracker::index_tracker_detail::Interval< T > Struct Template Reference

Interval object represents a range.

`#include <index_tracker.h>`

Public Member Functions

- **Interval** ()
  Default constructor.
- **Interval** (T a)
  Construct an interval from a to a+1.
- **Interval** (T a, T b)
  Construct an interval from a to b.
- **Interval** (const Interval< T > &rhs)
  Copy constructor.
- Interval & operator= (const Interval &rhs)
  Assigment operator overload.
- bool **has** (T x)
  Is x in the bounds of the interval.
- T **lower** () const
  Get the lower inclusive bound of the interval.
- T **upper** () const
  Get the upper exclusive bound of the interval.
- T & **lower** ()
  Get the lower inclusive bound of the interval.
- T & **upper** ()
  Get the upper exclusive bound of the interval.
- std::size_t **size** ()
  Get the size of the interval.

9.5.1 Detailed Description

template<typename T>
struct index_tracker::index_tracker_detail::Interval< T >

Template Parameters

T	Typename of the interval data

9.5.2 Member Function Documentation

9.5.2.1 operator=()

```
template<typename T >
Interval & index_tracker::index_tracker_detail::Interval< T >::operator= (
            const Interval< T > & rhs )  [inline]
```

Parameters

in	*rhs*	The right hand side

Returns

Reference to this

The documentation for this struct was generated from the following file:

- include/casc/index_tracker.h

9.6 casc::Orientable Struct Reference

Class representing the orientation.

`#include <Orientable.h>`

Data Fields

- int **orientation**

  Integer representing +/- 1 orientation.

The documentation for this struct was generated from the following file:

- include/casc/Orientable.h

9.7 util::range< T > Struct Template Reference

A range object to support range based for loops.

`#include <util.h>`

Public Member Functions

- template<class C >
  range (C &&c)

  Construct a range for a container class.
- range (T b, T e)

  Construct a range from an iterator.
- T begin ()

  Get the beginning iterator.
- T end ()

  Get the end iterator.

9.7.1 Detailed Description

template<typename T>
struct util::range< T >

This is a basic data structure which implements a `begin()` and `end()` functions for range based for lopping added in C++11. See also `range-for`.

Template Parameters

T	Typename of the iterator

9.7.2 Constructor & Destructor Documentation

9.7.2.1 range() [1/2]

```
template<typename T >
template<class C >
util::range< T >::range (
            C && c )  [inline]
```

Parameters

in	*c*	Container class which implements begin() and end().

Template Parameters

C	Typename of the container.

9.7.2.2 range() [2/2]

```
template<typename T >
util::range< T >::range (
            T b,
            T e )  [inline]
```

Parameters

in	*b*	Beginning iterator
in	*e*	End iterator.

9.7.3 Member Function Documentation

9.7.3.1 begin()

```
template<typename T >
T util::range< T >::begin ( )  [inline]
```

Returns

Returns an iterator to the beginning.

9.7.3.2 end()

```
template<typename T >
T util::range< T >::end ( )  [inline]
```

Returns

Returns an iterator to the end.

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.8 util::remove_first_val< Integer, IntegerSequence > Struct Template Reference

General template for removing the first value from a type holder.

```
#include <util.h>
```

9.8.1 Detailed Description

template<class Integer, class IntegerSequence>
struct util::remove_first_val< Integer, IntegerSequence >

Template Parameters

Integer	Typename of integer.
IntegerSequence	Sequence of compile time integers.

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.9 util::remove_first_val< Integer, InHolder< Integer, I, Is... > > Struct Template Reference

Specialization for removing first integer from a sequence of compile time integers.

`#include <util.h>`

Public Types

- using **type** = InHolder< Integer, Is... >
  Type holder with first value removed.

9.9.1 Detailed Description

template<class Integer, template< class, Integer... > class InHolder, Integer I, Integer... Is>
struct util::remove_first_val< Integer, InHolder< Integer, I, Is... > >

Template Parameters

Integer	Typename of integer type.
InHolder	Type holder of integer sequence.
I	The first integer
Is	Remaining integers

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.10 util::reverse_sequence< Integer, IntegerSequence > Struct Template Reference

Reverse an Integer Sequence.

`#include <util.h>`

Public Types

- using **type** = typename detail::reverse_sequence_helper< Integer, IntegerSequence >::type
  Reversed sequence of types.

9.10.1 Detailed Description

template<class Integer, class IntegerSequence>
struct util::reverse_sequence< Integer, IntegerSequence >

Template Parameters

Integer	Typename of an integer class.
IntegerSequence	Sequence of compile-time integers.

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.11 casc::simplicial_complex< traits >::SimplexID< k > Struct Template Reference

A handle for a simplex object in the complex.

`#include <SimplicialComplex.h>`

Public Types

- using **complex** = simplicial_complex< traits >
  Typename of the complex.

Public Member Functions

- **SimplexID** ()
  Default constructor wraps a nullptr.
- SimplexID (NodePtr< k > p)
  Constructor to wrap a NodePtr<k>.
- SimplexID (const SimplexID &rhs)
  Copy constructor.
- SimplexID & **operator=** (const SimplexID &rhs)
  Assignment operator.
- **operator std::uintptr_t** () const
  Support casting to uintptr_t for hashing.
- complex::NodeData< k > const & **operator*** () const
  Dereferencing a SimplexID returns the data stored.
- complex::NodeData< k > & **operator*** ()
  Dereferencing a SimplexID returns the data stored.
- complex::NodeData< k > const & **data** () const
  Get a handle to the stored data.
- complex::NodeData< k > & **data** ()
  Get a handle to the stored data.

- std::array< KeyType, k > indices () const

  Gets the name of a simplex as an std::Array.
- template<class Inserter >
  void cover_insert (Inserter pos) const

  Insert the coboundary keys of a simple into an inserter.
- std::vector< KeyType > cover () const

  Get the coboundary keys of a simplex.
- template<std::size_t j>
  SimplexID< k+j > get_simplex_up (const KeyType(&s)[j]) const

  Get a coboundary simplex.
- template<std::size_t j>
  SimplexID< k+j > get_simplex_up (const std::array< KeyType, j > &arr) const

  Get a coboundary simplex.
- SimplexID< k+1 > get_simplex_up (const KeyType s) const

  Convenience version of get_simplex_up when the name 's' consists of a single character.
- template<std::size_t j>
  SimplexID< k-j > **get_simplex_down** (const KeyType(&s)[j]) const

  Gets the simplex down.
- template<std::size_t j>
  SimplexID< k-j > **get_simplex_down** (const std::array< KeyType, j > &arr) const

  Gets the simplex down.
- SimplexID< k-1 > **get_simplex_down** (const KeyType s) const

  Gets the simplex down.

Data Fields

- friend **simplicial_complex**< **traits** >

  SimplexID is a friend of the complex.

Static Public Attributes

- static constexpr std::size_t **level** = k

  The dimension of the simplex.

Friends

- bool **operator==** (SimplexID lhs, SimplexID rhs)

  Equality of wrapped pointers.
- bool **operator!=** (SimplexID lhs, SimplexID rhs)

  Inequality of wrapped pointers.
- bool **operator<=** (SimplexID lhs, SimplexID rhs)

  Compare wrapped pointers.
- bool **operator>=** (SimplexID lhs, SimplexID rhs)

  Compare wrapped pointers.
- bool **operator<** (SimplexID lhs, SimplexID rhs)

  Compare wrapped pointers.
- bool **operator>** (SimplexID lhs, SimplexID rhs)

  Compare wrapped pointers.
- std::ostream & operator<< (std::ostream &out, const SimplexID &nid)

  Print the simplex as its name.

9.11.1 Detailed Description

template<typename traits>
template<std::size_t k>
struct casc::simplicial_complex< traits >::SimplexID< k >

SimplexID wraps a Node* for external handling. This way the end users are never exposed to a raw pointer. For all general purposes algorithms should use and pass SimplexIDs over raw pointers.

Template Parameters

k	The Simplex dimension.

9.11.2 Constructor & Destructor Documentation

9.11.2.1 SimplexID() [1/2]

```
template<typename traits >
template<std::size_t k>
casc::simplicial_complex< traits >::SimplexID< k >::SimplexID (
            NodePtr< k > p )  [inline]
```

Parameters

in	*p*	The NodePtr to wrap

9.11.2.2 SimplexID() [2/2]

```
template<typename traits >
template<std::size_t k>
casc::simplicial_complex< traits >::SimplexID< k >::SimplexID (
            const SimplexID< k > & rhs )  [inline]
```

Parameters

in	*rhs*	Another SimplexID to copy.

9.11.3 Member Function Documentation

9.11.3.1 cover()

```
template<typename traits >
template<std::size_t k>
std::vector< KeyType > casc::simplicial_complex< traits >::SimplexID< k >::cover ( ) const
[inline]
```

Returns

A vector of coboundary indices.

9.11.3.2 cover_insert()

```
template<typename traits >
template<std::size_t k>
template<class Inserter >
void casc::simplicial_complex< traits >::SimplexID< k >::cover_insert (
            Inserter pos ) const  [inline]
```

Parameters

in	*pos*	Iterator inserter

Template Parameters

Inserter	Typename of the inserter.

9.11.3.3 get_simplex_up() [1/3]

```
template<typename traits >
template<std::size_t k>
SimplexID< k+1 > casc::simplicial_complex< traits >::SimplexID< k >::get_simplex_up (
            const KeyType s ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*s*	The relative single character name of the desired simplex.

Template Parameters

i	The size of simplex 'id'.

Returns

SimplexID of node corresponding to $id \cup s$.

9.11.3.4 get_simplex_up() [2/3]

```
template<typename traits >
template<std::size_t k>
template<std::size_t j>
SimplexID< k+j > casc::simplicial_complex< traits >::SimplexID< k >::get_simplex_up (
            const KeyType(&) s[j] ) const  [inline]
```

Parameters

in	*s*	Array of keys to follow

Template Parameters

j	Number of keys

Returns

The simplex up

9.11.3.5 get_simplex_up() [3/3]

```
template<typename traits >
template<std::size_t k>
template<std::size_t j>
SimplexID< k+j > casc::simplicial_complex< traits >::SimplexID< k >::get_simplex_up (
            const std::array< KeyType, j > & arr ) const  [inline]
```

Parameters

in	*arr*	Array of keys to follow

Template Parameters

j	Number of keys

Returns

The simplex up

9.11.3.6 indices()

```
template<typename traits >
template<std::size_t k>
std::array< KeyType, k > casc::simplicial_complex< traits >::SimplexID< k >::indices ( )
const  [inline]
```

Parameters

in	*id*	SimplexID of the simplex of interest.

Returns

Array containing the name of 'id'.

9.11.4 Friends And Related Function Documentation

9.11.4.1 operator<<

```
template<typename traits >
template<std::size_t k>
std::ostream & operator<< (
            std::ostream & out,
            const SimplexID< k > & nid )  [friend]
```

Parameters

	out	Handle to the stream
in	*nid*	SimplexID of interest

Returns

Handle to the stream

Example

```
 {(.c)}
mesh.insert<3>({0,1,2});
std::cout « s « std::endl;
s{0,1,2}"
```

The documentation for this struct was generated from the following file:

- include/casc/SimplicialComplex.h

9.12 casc::SimplexMap< Complex > Struct Template Reference

A multimap to represent a map of simplex indices to a set of simplices.

`#include <SimplexMap.h>`

Public Types

- template<std::size_t j>
  using **SimplexID** = typename Complex::template SimplexID< j >

  Alias for SimplexID.
- using **LevelIndex** = typename Complex::LevelIndex

  Index sequence of types from the simplicial_complex.
- using **cLevelIndex** = typename util::remove_first_val< std::size_t, LevelIndex >::type

  Index sequence starting at 1.
- using **RevIndex** = typename util::reverse_sequence< std::size_t, LevelIndex >::type

  Reversed Index sequence.
- using **cRevIndex** = typename util::reverse_sequence< std::size_t, cLevelIndex >::type

  Reversed index sequence stops at 1.
- using **type_this** = SimplexMap< Complex >

  Typename of this object.

Public Member Functions

- **SimplexMap** ()

  Default constructor.
- template<std::size_t k>
  auto & get ()

  Get the map for a particular simplex dimension.
- template<std::size_t k>
  auto & get () const

Friends

- std::ostream & operator<< (std::ostream &output, const SimplexMap< Complex > &S)

  Print the SimplexMap.

9.12.1 Detailed Description

template<typename Complex>
struct casc::SimplexMap< Complex >

Template Parameters

Complex	Typename of the simplicial_complex.

9.12.2 Member Function Documentation

9.12.2.1 get() [1/2]

```
template<typename Complex >
template<std::size_t k>
```

```
auto & casc::SimplexMap< Complex >::get ( )  [inline]
```

Template Parameters

k	Simplex dimension to retrieve.

Returns

A map of SimplexID<k> to SimplexSet.

9.12.2.2 get() [2/2]

```
template<typename Complex >
template<std::size_t k>
auto & casc::SimplexMap< Complex >::get ( ) const  [inline]
```

This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

9.12.3 Friends And Related Function Documentation

9.12.3.1 operator<<

```
template<typename Complex >
std::ostream & operator<< (
            std::ostream & output,
            const SimplexMap< Complex > & S )  [friend]
```

Parameters

	output	Handle to the stream to print to.
in	*S*	SimplexMap to print.

Returns

Handle to the stream.

The documentation for this struct was generated from the following file:

- include/casc/SimplexMap.h

9.13 casc::SimplexSet< Complex > Struct Template Reference

A multiset to store simplices in a simplicial_complex.

`#include <SimplexSet.h>`

Public Types

- template<std::size_t j>
  using **SimplexID** = typename Complex::template SimplexID< j >
  Alias for SimplexID.
- using **LevelIndex** = typename Complex::LevelIndex
  Index sequence of types from the simplicial_complex.
- using **cLevelIndex** = typename util::remove_first_val< std::size_t, LevelIndex >::type
  Index sequence starting at 1.
- using **RevIndex** = typename util::reverse_sequence< std::size_t, LevelIndex >::type
  Reversed index sequence.
- using **cRevIndex** = typename util::reverse_sequence< std::size_t, cLevelIndex >::type
  Reversed index sequence stops at 1.
- using **type_this** = SimplexSet< Complex >
  Typename of this.
- using **SimplexIDLevel** = typename util::int_type_map< std::size_t, std::tuple, LevelIndex, SimplexID >::type
  Tuple of SimplexIDs wrt an integral level.

Public Member Functions

- **SimplexSet** ()
  Default constructor.
- ~**SimplexSet** ()
  Default destructor.
- template<std::size_t k>
  auto empty () const noexcept
  Checks if a level has no elements.
- template<std::size_t k>
  auto size () const noexcept
  Return the number of elements in a level.
- void **clear** ()
  Clear the contents.
- template<std::size_t k>
  void insert (SimplexID< k > s)
  Insert a simplex into the set.
- void insert (const SimplexSet< Complex > &s)
  Insert a SimplexSet into this.
- template<std::size_t k>
  void erase (SimplexID< k > s)
  Remove a simplex from the set.
- void erase (const SimplexSet< Complex > &s)
  Remove a set of simplices.
- template<std::size_t k>
  auto find (const SimplexID< k > s)

Get the simplex of interest.

- template<std::size_t k>
  auto find (const SimplexID< k > s) const

  Get the simplex of interest.
- template<std::size_t k>
  auto end ()

  Get the past-the-end iterator.
- template<std::size_t k>
  auto cend () const

  Get the past-the-end iterator.
- template<std::size_t k>
  auto begin ()

  Get an interator to the first element of the container.
- template<std::size_t k>
  auto cbegin () const

  Get an interator to the first element of the container.
- template<std::size_t k>
  auto & **get** ()
- template<std::size_t k>
  auto & **get** () const

Data Fields

- util::type_map< SimplexIDLevel, NodeSet >::type **tupleSet**

  Tuple of NodeSets per level.

Friends

- std::ostream & operator<< (std::ostream &output, const SimplexSet< Complex > &S)

  Print the SimplexSet.

9.13.1 Detailed Description

template<typename Complex>
struct casc::SimplexSet< Complex >

This is really a tuple of sets where each set corresponds to a simplex dimension. Many convenience functions are wrapped so this behaves much like a std::set.

Template Parameters

Complex	Typename of the simplicial_complex.

9.13.2 Member Function Documentation

9.13.2.1 begin()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::begin ( )  [inline]
```

Template Parameters

k	The simplex dimension to get iterator of.

Returns

Returns an iterator to the first element.

9.13.2.2 cbegin()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::cbegin ( ) const  [inline]
```

Template Parameters

k	The simplex dimension to get iterator of.

Returns

Returns an iterator to the first element.

9.13.2.3 cend()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::cend ( ) const  [inline]
```

Template Parameters

k	The simplex dimension to get iterator of.

Returns

Returns an iterator to the element following the last element of the set for the specified simplex dimension.

9.13.2.4 empty()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::empty ( ) const  [inline], [noexcept]
```

Template Parameters

k	Level to check.

Returns

True if the container is empty, false otherwise.

9.13.2.5 end()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::end ( )  [inline]
```

Template Parameters

k	The simplex dimension to get iterator of.

Returns

Returns an iterator to the element following the last element of the set for the specified simplex dimension.

9.13.2.6 erase() [1/2]

```
template<typename Complex >
void casc::SimplexSet< Complex >::erase (
            const SimplexSet< Complex > & s )  [inline]
```

Parameters

in	*s*	SimplexSet to remove.

9.13.2.7 erase() [2/2]

```
template<typename Complex >
template<std::size_t k>
```

```
void casc::SimplexSet< Complex >::erase (
            SimplexID< k > s )  [inline]
```

Parameters

in	*s*	Simplex to remove.

Template Parameters

k	Simplex dimension of 's'.

9.13.2.8 find() [1/2]

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::find (
            const SimplexID< k > s )  [inline]
```

Parameters

in	*s*	The simplex to search for.

Template Parameters

k	Simplex dimension of 's'.

Returns

Iterator to an element with key equivalent to s. If no such element is found, past-the-end iterator (see end()) is returned.

9.13.2.9 find() [2/2]

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::find (
            const SimplexID< k > s ) const  [inline]
```

Parameters

in	*s*	The simplex to search for.

Template Parameters

k	Simplex dimension of 's'.

Returns

Iterator to an element with key equivalent to s. If no such element is found, past-the-end iterator (see end()) is returned.

9.13.2.10 insert() [1/2]

```
template<typename Complex >
void casc::SimplexSet< Complex >::insert (
            const SimplexSet< Complex > & s )  [inline]
```

Parameters

in	*s*	The SimplexSet to insert.

9.13.2.11 insert() [2/2]

```
template<typename Complex >
template<std::size_t k>
void casc::SimplexSet< Complex >::insert (
            SimplexID< k > s )  [inline]
```

Parameters

in	*s*	Simplex to insert.

Template Parameters

k	Simplex dimension of 's'.

9.13.2.12 size()

```
template<typename Complex >
template<std::size_t k>
auto casc::SimplexSet< Complex >::size ( ) const  [inline], [noexcept]
```

Template Parameters

k	Simplex dimension to query

Returns

Returns the number of simplices of dimension `k` are in the set.

9.13.3 Friends And Related Function Documentation

9.13.3.1 operator<<

```
template<typename Complex >
std::ostream & operator<< (
            std::ostream & output,
            const SimplexSet< Complex > & S )  [friend]
```

See also casc::simplicial_complex::SimplexID::operator<<.

Parameters

	output	Handle to the stream to print to.
in	*S*	SimplexSet to print.

Returns

Handle to the stream.

The documentation for this struct was generated from the following file:

- include/casc/SimplexSet.h

9.14 casc::simplicial_complex< traits > Class Template Reference

The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring.

`#include <SimplicialComplex.h>`

Data Structures

- struct EdgeID

  External reference to an edge or a connection within the complex.
- struct SimplexID

  A handle for a simplex object in the complex.

Public Types

- using **KeyType** = typename traits::KeyType

  Typename of simplex keys.
- using **NodeDataTypes** = typename traits::NodeTypes

  Typenames of the data stored on simplices.
- using **EdgeDataTypes** = typename traits::EdgeTypes

  Typenames of the data stored on edges.
- using **type_this** = simplicial_complex< traits >

  Type of this.
- using **LevelIndex** = typename std::make_index_sequence< numLevels >

  Index of all simplex dimensions in the complex.
- template<std::size_t k>
  using NodeData = typename util::type_get< k, NodeDataTypes >::type
- template<std::size_t k>
  using EdgeData = typename util::type_get< k, EdgeDataTypes >::type

Public Member Functions

- **simplicial_complex** ()

  Default constructor.
- ~simplicial_complex ()

  Destruct the simplicial complex.
- template<std::size_t n>
  SimplexID< n > insert (const KeyType(&s)[n])

  Insert a simplex and all sub-simplices into the complex.
- template<std::size_t n>
  SimplexID< n > insert (const KeyType(&s)[n], const NodeData< n > &data)

  Insert a simplex and all sub-simplices into the complex along with data.
- template<std::size_t n>
  SimplexID< n > insert (const std::array< KeyType, n > &s)

  Insert a simplex named and all sub-simplices into the complex.
- template<std::size_t n>
  SimplexID< n > insert (const std::array< KeyType, n > &s, const NodeData< n > &data)

  Insert a simplex and all sub-simplices into the complex along with data.
- KeyType add_vertex ()

  Add a new vertex to the complex.
- KeyType add_vertex (const NodeData< 1 > &data)

  Add a new vertex to the complex with data.
- template<std::size_t n, typename Lambda >
  void get_name (SimplexID< n > id, Lambda fn) const

  Apply a lambda function the name of a simplex.
- template<std::size_t n>
  std::array< KeyType, n > get_name (SimplexID< n > id) const

  Gets the name of a simplex as an std::Array.
- std::array< KeyType, 0 > get_name (SimplexID< 0 >) const

  Gets the name of a simplex.
- template<std::size_t n>
  SimplexID< n > get_simplex_up (const KeyType(&s)[n]) const

  Gets the simplex with name 's'.
- template<std::size_t n>
  SimplexID< n > **get_simplex_up** (const std::array< KeyType, n > &arr) const

- template<std::size_t i, std::size_t j>
  SimplexID< i+j > get_simplex_up (const SimplexID< i > id, const KeyType(&s)[j]) const

  Get the simplex identifier which has the name 's' relative to the simplex 'id'.
- template<std::size_t i, std::size_t j>
  SimplexID< i+j > **get_simplex_up** (const SimplexID< i > id, const std::array< KeyType, j > &arr) const
- template<std::size_t i>
  SimplexID< i+1 > get_simplex_up (const SimplexID< i > id, const KeyType s) const

  Convenience version of get_simplex_up when the name 's' consists of a single character.
- SimplexID< 0 > get_simplex_up () const

  Get the root simplex.
- template<std::size_t i, std::size_t j>
  SimplexID< i-j > get_simplex_down (const SimplexID< i > id, const KeyType(&s)[j]) const

  Get the sub-simplex of the simplex 'id' which does not have 's' in the name.
- template<std::size_t i, std::size_t j>
  SimplexID< i-j > **get_simplex_down** (const SimplexID< i > id, const std::array< KeyType, j > &arr) const
- template<std::size_t i>
  SimplexID< i-1 > get_simplex_down (const SimplexID< i > id, const KeyType s) const

  Convenience version of get_simplex_down when the name 's' consists of a single character.
- SimplexID< 0 > get_simplex_down () const

  Get the root simplex.
- template<std::size_t k, class Inserter >
  void get_cover_insert (const SimplexID< k > id, Inserter pos) const

  Insert the coboundary keys of a simple into an inserter.
- template<std::size_t k, class Lambda >
  void get_cover (const SimplexID< k > id, Lambda fn) const

  Apply a lambda function to the coboundary keys.
- template<std::size_t k>
  std::vector< KeyType > get_cover (const SimplexID< k > id) const

  Get the coboundary keys of a simplex.
- template<std::size_t k>
  std::set< SimplexID< k+1 > > up (const std::set< SimplexID< k > > &&simplices) const

  Get the coboundary of a set of simplices.
- template<std::size_t k>
  std::set< SimplexID< k+1 > > up (const std::set< SimplexID< k > > &simplices) const

  Get the coboundary of a set of simplices.
- template<std::size_t k>
  std::set< SimplexID< k+1 > > up (const SimplexID< k > nid) const

  Get the coboundary of a simplex.
- template<std::size_t k, class InsertIter >
  void **up** (const std::set< SimplexID< k > > &&simplices, InsertIter iter) const
- template<std::size_t k, class InsertIter >
  void **up** (const std::set< SimplexID< k > > &simplices, InsertIter iter) const
- template<std::size_t k, class InsertIter >
  void **up** (const SimplexID< k > simplex, InsertIter iter) const
- template<std::size_t k>
  std::set< SimplexID< k-1 > > down (const std::set< SimplexID< k > > &&simplices) const

  Get the boundary of a set of simplices.
- template<std::size_t k>
  std::set< SimplexID< k-1 > > down (const std::set< SimplexID< k > > &simplices) const

  Get the boundary of a set of simplices.
- template<std::size_t k>
  std::set< SimplexID< k-1 > > down (const SimplexID< k > simplex) const

  Get the boundary of a simplex.

- template<std::size_t k, class InsertIter >
  void **down** (const std::set< SimplexID< k > > &&simplices, InsertIter iter) const
- template<std::size_t k, class InsertIter >
  void **down** (const std::set< SimplexID< k > > &simplices, InsertIter iter) const
- template<std::size_t k, class InsertIter >
  void **down** (const SimplexID< k > simplex, InsertIter iter) const
- template<std::size_t k>
  EdgeID< k+1 > get_edge_up (SimplexID< k > simplex, KeyType a)

  Gets the edge up from a simplex.
- template<std::size_t k>
  EdgeID< k > get_edge_down (SimplexID< k > simplex, KeyType a)

  Gets the edge down from a simplex.
- template<std::size_t k>
  EdgeID< k+1 > get_edge_up (SimplexID< k > simplex, KeyType a) const

  Gets the edge up from a simplex.
- template<std::size_t k>
  EdgeID< k > get_edge_down (SimplexID< k > simplex, KeyType a) const

  Gets the edge down from a simplex.
- template<std::size_t k>
  bool exists (const KeyType(&s)[k]) const

  Check whether a simplex with some name exists.
- template<std::size_t k>
  std::size_t size () const

  Get the number of simplices of dimension 'k'.
- template<std::size_t k>
  auto get_level_id ()

  Create an iterator to traverse the SimplexIDs of a dimension.
- template<std::size_t k>
  auto get_level_id () const

  Create an iterator to traverse the SimplexIDs of a dimension.
- template<std::size_t k>
  auto get_level ()

  Create an iterator to traverse the simplex data of a dimension.
- template<std::size_t k>
  auto get_level () const

  Create an iterator to traverse the simplex data of a dimension.
- template<std::size_t k>
  std::size_t remove (const KeyType(&s)[k])

  Remove a simplex and all dependent simplices by name.
- template<std::size_t k>
  std::size_t remove (const std::array< KeyType, k > &s)

  Remove a simplex and all dependent simplices by name.
- template<std::size_t k>
  std::size_t remove (SimplexID< k > s)

  Remove a simplex and all dependent simplices by SimplexID.
- template<std::size_t k>
  bool onBoundary (const SimplexID< k > s) const

  Checks whether a simplex is on a boundary.
- template<std::size_t level>
  bool nearBoundary (const SimplexID< level > s) const

  Checks whether a simplex is near a boundary.
- template<std::size_t L, std::size_t R>
  bool leq (SimplexID< L > lhs, SimplexID< R > rhs) const

Less than or equal to comparison operator of two SimplexIDs.

- template<std::size_t L, std::size_t R>
  bool eq (SimplexID< L >, SimplexID< R >) const

  Equality comparison of two simplices.
- template<std::size_t k>
  bool eq (SimplexID< k > lhs, SimplexID< k > rhs) const

  Equality comparison of two simplices.
- template<std::size_t L, std::size_t R>
  bool lt (SimplexID< L > lhs, SimplexID< R > rhs) const

  Less than comparison of simplices.

Static Public Attributes

- static constexpr std::size_t **numLevels** = NodeDataTypes::size

  Total number of levels in the complex.
- static constexpr std::size_t **topLevel** = numLevels-1

  Dimension of the simplicial complex.
- static constexpr std::size_t **bdryLevel** = numLevels-2

  Dimension of boundaries.

Friends

- struct SimplexID
- struct EdgeID

9.14.1 Detailed Description

template<typename traits>
class casc::simplicial_complex< traits >

You can create a CASC object by defining a struct containing the traits of the complex. For example:

```
struct complex_traits{
    using KeyType = int;
    using NodeTypes = util::type_holder<int,int,int,int>;
    using EdgeTypes = util::type_holder<int,int,int>;
};

using SurfaceMesh = simplicial_complex<complex_traits>;
```

This is the preferred method for creating a new CASC type. Alternatively you can use the AbstractSimplicialComplex alias to build a struct for you.

Template Parameters

traits	A struct defining the dimension of the complex and data to be stored on each node and edge.

9.14.2 Member Typedef Documentation

9.14.2.1 EdgeData

```
template<typename traits >
template<std::size_t k>
using casc::simplicial_complex< traits >::EdgeData = typename util::type_get<k, EdgeDataTypes>↩
::type
```

Convenience alias for the user specified EdgeData<k> typename

9.14.2.2 NodeData

```
template<typename traits >
template<std::size_t k>
using casc::simplicial_complex< traits >::NodeData = typename util::type_get<k, NodeDataTypes>↩
::type
```

Convenience alias for the user specified NodeData<k> typename

9.14.3 Constructor & Destructor Documentation

9.14.3.1 ~simplicial_complex()

```
template<typename traits >
casc::simplicial_complex< traits >::~simplicial_complex ( )  [inline]
```

Recursively go over the simplices and remove them prior to destructing the CASC object itself.

9.14.4 Member Function Documentation

9.14.4.1 add_vertex() [1/2]

```
template<typename traits >
KeyType casc::simplicial_complex< traits >::add_vertex ( )  [inline]
```

A list of currently unused indices are tracked using a B-tree. This function retrieves a currently unused index and creates a new vertex while returning the new key.

Returns

The key of the new vertex.

9.14.4.2 add_vertex() [2/2]

```
template<typename traits >
KeyType casc::simplicial_complex< traits >::add_vertex (
            const NodeData< 1 > & data )  [inline]
```

Returns

The key of the new vertex.

9.14.4.3 down() [1/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k-1 > > casc::simplicial_complex< traits >::down (
            const SimplexID< k > simplex ) const  [inline]
```

Parameters

simplex	The simplex of interest.

Template Parameters

k	The dimension of the simplex.

Returns

Set of (k-1)-simplices of which 'simplex' is a coface of.

9.14.4.4 down() [2/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k-1 > > casc::simplicial_complex< traits >::down (
            const std::set< SimplexID< k > > && simplices ) const  [inline]
```

Parameters

simplices	The set of simplicies.

Template Parameters

k	The dimension of the simplices.

Returns

The set of boundary simplices.

9.14.4.5 down() [3/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k-1 > > casc::simplicial_complex< traits >::down (
            const std::set< SimplexID< k > > & simplices ) const  [inline]
```

Parameters

simplices	The set of simplicies.

Template Parameters

k	The dimension of the simplices.

Returns

The set of boundary simplices.

9.14.4.6 eq() [1/2]

```
template<typename traits >
template<std::size_t k>
bool casc::simplicial_complex< traits >::eq (
            SimplexID< k > lhs,
            SimplexID< k > rhs ) const  [inline]
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

k	Dimension of the simplices.

Returns

True if the names are the same.

9.14.4.7 eq() [2/2]

```
template<typename traits >
template<std::size_t L, std::size_t R>
bool casc::simplicial_complex< traits >::eq (
            SimplexID< L > ,
            SimplexID< R >  ) const  [inline]
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

L	Dimension of lhs simplex.
R	Dimension of rhs simplex.

Returns

Always false as L != R. The L==R case is overloaded by partial specialization.

9.14.4.8 exists()

```
template<typename traits >
template<std::size_t k>
bool casc::simplicial_complex< traits >::exists (
            const KeyType(&) s[k] ) const  [inline]
```

Parameters

in	*s*	C-style array of the name

Template Parameters

k	The dimension of the simplex.

Returns

True if the simplex is in the complex.

9.14.4.9 get_cover() [1/2]

```
template<typename traits >
template<std::size_t k>
```

```
std::vector< KeyType > casc::simplicial_complex< traits >::get_cover (
            const SimplexID< k > id ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.

Template Parameters

k	The dimension of the simplex.

Returns

A vector of coboundary indices.

9.14.4.10 get_cover() [2/2]

```
template<typename traits >
template<std::size_t k, class Lambda >
void casc::simplicial_complex< traits >::get_cover (
            const SimplexID< k > id,
            Lambda fn ) const  [inline]
```

Parameters

in	*id*	The identifier
in	*fn*	The function

Template Parameters

k	The dimension of the simplex.
Lambda	Typename of a functor which supports operator(KeyType).

9.14.4.11 get_cover_insert()

```
template<typename traits >
template<std::size_t k, class Inserter >
void casc::simplicial_complex< traits >::get_cover_insert (
            const SimplexID< k > id,
            Inserter pos ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*pos*	Iterator inserter

Template Parameters

k	The dimension of the simplex.
Inserter	Typename of the inserter.

9.14.4.12 get_edge_down() [1/2]

```
template<typename traits >
template<std::size_t k>
EdgeID< k > casc::simplicial_complex< traits >::get_edge_down (
            SimplexID< k > simplex,
            KeyType a )  [inline]
```

Parameters

in	*simplex*	The simplex of interest.
in	*a*	Key of the edge to get.

Template Parameters

k	The level of the simplex of interest

Returns

The edge down.

9.14.4.13 get_edge_down() [2/2]

```
template<typename traits >
template<std::size_t k>
EdgeID< k > casc::simplicial_complex< traits >::get_edge_down (
            SimplexID< k > simplex,
            KeyType a ) const  [inline]
```

Parameters

in	*simplex*	The simplex of interest.
in	*a*	Key of the edge to get.

Template Parameters

k	The level of the simplex of interest

Returns

The edge down.

9.14.4.14 get_edge_up() [1/2]

```
template<typename traits >
template<std::size_t k>
EdgeID< k+1 > casc::simplicial_complex< traits >::get_edge_up (
            SimplexID< k > simplex,
            KeyType a )  [inline]
```

Parameters

in	*simplex*	The simplex of interest.
in	*a*	Key of the edge to get.

Template Parameters

k	The level of the simplex of interest

Returns

The edge up.

9.14.4.15 get_edge_up() [2/2]

```
template<typename traits >
template<std::size_t k>
EdgeID< k+1 > casc::simplicial_complex< traits >::get_edge_up (
            SimplexID< k > simplex,
            KeyType a ) const  [inline]
```

Parameters

in	*simplex*	The simplex of interest.
in	*a*	Key of the edge to get.

Template Parameters

k	The level of the simplex of interest

Returns

The edge up.

9.14.4.16 get_level() [1/2]

```
template<typename traits >
template<std::size_t k>
auto casc::simplicial_complex< traits >::get_level ( )  [inline]
```

Template Parameters

k	The simplex dimension to traverse.

Returns

An iterator across the data of all k-simplices in the complex.

9.14.4.17 get_level() [2/2]

```
template<typename traits >
template<std::size_t k>
auto casc::simplicial_complex< traits >::get_level ( ) const  [inline]
```

Template Parameters

k	The simplex dimension to traverse.

Returns

An iterator across the data of all k-simplices in the complex.

9.14.4.18 get_level_id() [1/2]

```
template<typename traits >
template<std::size_t k>
auto casc::simplicial_complex< traits >::get_level_id ( )  [inline]
```

Template Parameters

k	The simplex dimension to traverse.

Returns

An iterator across all k-simplices of the complex.

9.14.4.19 get_level_id() [2/2]

```
template<typename traits >
template<std::size_t k>
auto casc::simplicial_complex< traits >::get_level_id ( ) const  [inline]
```

Template Parameters

k	The simplex dimension to traverse.

Returns

An iterator across all k-simplices of the complex.

9.14.4.20 get_name() [1/3]

```
template<typename traits >
std::array< KeyType, 0 > casc::simplicial_complex< traits >::get_name (
            SimplexID< 0 >  ) const  [inline]
```

This is the explicit specialization which handles the empty set simplex.

Parameters

in	*id*	SimplexID of the simplex of interest.

Returns

Array containing the name of 'id'.

9.14.4.21 get_name() [2/3]

```
template<typename traits >
template<std::size_t n>
std::array< KeyType, n > casc::simplicial_complex< traits >::get_name (
            SimplexID< n > id ) const  [inline]
```

Parameters

in	*id*	SimplexID of the simplex of interest.

Template Parameters

n	Size of the simplex referenced by 'id'.

Returns

Array containing the name of 'id'.

9.14.4.22 get_name() [3/3]

```
template<typename traits >
template<std::size_t n, typename Lambda >
void casc::simplicial_complex< traits >::get_name (
            SimplexID< n > id,
            Lambda fn ) const  [inline]
```

Parameters

in	*id*	SimplexID of the simplex of interest.
in	*fn*	Lambda function to apply to the name of 'id'.

Template Parameters

n	Dimension of simplex 'id'.
Lambda	Functor which supports operator(KeyType).

9.14.4.23 get_simplex_down() [1/3]

```
template<typename traits >
SimplexID< 0 > casc::simplicial_complex< traits >::get_simplex_down ( ) const  [inline]
```

Returns

The root simplex.

9.14.4.24 get_simplex_down() [2/3]

```
template<typename traits >
template<std::size_t i>
SimplexID< i-1 > casc::simplicial_complex< traits >::get_simplex_down (
            const SimplexID< i > id,
            const KeyType s ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*s*	The relative single character name of the desired simplex.

Template Parameters

i	The size of simplex 'id'.

Returns

The node down.

9.14.4.25 get_simplex_down() [3/3]

```
template<typename traits >
template<std::size_t i, std::size_t j>
SimplexID< i-j > casc::simplicial_complex< traits >::get_simplex_down (
            const SimplexID< i > id,
            const KeyType(&) s[j] ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*s*	The relative name of the desired simplex.

Template Parameters

i	The size of simplex 'id'.
j	The length of the name 's'

Returns

The node down.

9.14.4.26 get_simplex_up() [1/4]

```
template<typename traits >
SimplexID< 0 > casc::simplicial_complex< traits >::get_simplex_up ( ) const  [inline]
```

Returns

The root simplex.

9.14.4.27 get_simplex_up() [2/4]

```
template<typename traits >
template<std::size_t n>
SimplexID< n > casc::simplicial_complex< traits >::get_simplex_up (
            const KeyType(&) s[n] ) const  [inline]
```

Parameters

in	*s*	Name of the simplex to find.

Template Parameters

n	Dimension of simplex s.

Returns

SimplexID of node corresponding to 's'.

9.14.4.28 get_simplex_up() [3/4]

```
template<typename traits >
template<std::size_t i>
SimplexID< i+1 > casc::simplicial_complex< traits >::get_simplex_up (
            const SimplexID< i > id,
            const KeyType s ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*s*	The relative single character name of the desired simplex.

Template Parameters

i	The size of simplex 'id'.

Returns

SimplexID of node corresponding to $id \cup s$.

9.14.4.29 get_simplex_up() [4/4]

```
template<typename traits >
template<std::size_t i, std::size_t j>
SimplexID< i+j > casc::simplicial_complex< traits >::get_simplex_up (
            const SimplexID< i > id,
            const KeyType(&) s[j] ) const  [inline]
```

Parameters

in	*id*	The identifier of a simplex.
in	*s*	The relative name of the desired simplex.

Template Parameters

i	The size of simplex 'id'.
j	The length of the name 's'.

Returns

SimplexID of node corresponding to $id \cup s$.

9.14.4.30 insert() [1/4]

```
template<typename traits >
template<std::size_t n>
SimplexID< n > casc::simplicial_complex< traits >::insert (
            const KeyType(&) s[n] )  [inline]
```

Example – insert the simplex {1,2,3}:
```
mesh.insert<3>({1,2,3});
```

Parameters

in	*s*	A C style array of vertices of simplex 's'.

Template Parameters

n	Dimension of simplex 's'.

9.14.4.31 insert() [2/4]

```
template<typename traits >
template<std::size_t n>
SimplexID< n > casc::simplicial_complex< traits >::insert (
            const KeyType(&) s[n],
            const NodeData< n > & data )  [inline]
```

Example – insert the simplex {1,2,3} with data:
`mesh.insert<3>({1,2,3}, 5);`

Parameters

in	*s*	A C style array of vertices of simplex 's'.
in	*data*	The data to be stored at the simplex 's'.

Template Parameters

n	Dimension of simplex 's'.

9.14.4.32 insert() [3/4]

```
template<typename traits >
template<std::size_t n>
SimplexID< n > casc::simplicial_complex< traits >::insert (
            const std::array< KeyType, n > & s )  [inline]
```

Parameters

in	*s*	Array of vertices comprising 's'.

Template Parameters

n	Dimension of simplex 's'.

9.14.4.33 insert() [4/4]

```
template<typename traits >
template<std::size_t n>
SimplexID< n > casc::simplicial_complex< traits >::insert (
            const std::array< KeyType, n > & s,
            const NodeData< n > & data )  [inline]
```

Parameters

in	*s*	Array of vertices comprising 's'.
in	*data*	The data to be stored at the simplex 's'.

Template Parameters

n	Dimension of simplex 's'.

9.14.4.34 leq()

```
template<typename traits >
template<std::size_t L, std::size_t R>
bool casc::simplicial_complex< traits >::leq (
            SimplexID< L > lhs,
            SimplexID< R > rhs ) const  [inline]
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

L	Dimension of lhs simplex.
R	Dimension of rhs simplex.

Returns

True if lhs is rhs or a proper face of rhs.

9.14.4.35 lt()

```
template<typename traits >
template<std::size_t L, std::size_t R>
bool casc::simplicial_complex< traits >::lt (
            SimplexID< L > lhs,
            SimplexID< R > rhs ) const  [inline]
```

Parameters

in	*lhs*	The left hand side
in	*rhs*	The right hand side

Template Parameters

L	Dimension of lhs simplex.
R	Dimension of rhs simplex.

Returns

True if lhs is a proper subface of rhs.

9.14.4.36 nearBoundary()

```
template<typename traits >
template<std::size_t level>
bool casc::simplicial_complex< traits >::nearBoundary (
            const SimplexID< level > s ) const  [inline]
```

Parameters

in	*s*	SimplexID of interest

Template Parameters

level	Dimension of the simplex

Returns

True if the simplex or any subsimplices are onBoundary.

9.14.4.37 onBoundary()

```
template<typename traits >
template<std::size_t k>
bool casc::simplicial_complex< traits >::onBoundary (
            const SimplexID< k > s ) const  [inline]
```

Parameters

in	*s*	SimplexID of interest

Template Parameters

k	Dimension of the simplex

Returns

True if the simplex is a member of a topLevel-1 simplex on the boundary or if the simplex is on a boundary or if the simplex is a coboundary of a boundary topLevel-1 simplex.

9.14.4.38 remove() [1/3]

```
template<typename traits >
template<std::size_t k>
std::size_t casc::simplicial_complex< traits >::remove (
            const KeyType(&) s[k] )  [inline]
```

Parameters

in	*s*	C-style array with the name of the simplex to remove.

Template Parameters

k	The dimension of the simplex.

Returns

Integer corresponding to the number of simplices removed.

9.14.4.39 remove() [2/3]

```
template<typename traits >
template<std::size_t k>
std::size_t casc::simplicial_complex< traits >::remove (
            const std::array< KeyType, k > & s )  [inline]
```

Parameters

in	*s*	std::array with the name of the simplex to remove.

Template Parameters

k	The dimension of the simplex.

Returns

Integer corresponding to the number of simplices removed.

9.14.4.40 remove() [3/3]

```
template<typename traits >
template<std::size_t k>
std::size_t casc::simplicial_complex< traits >::remove (
            SimplexID< k > s )  [inline]
```

Parameters

in	*s*	SimplexID of the simplex to remove.

Template Parameters

k	The dimension of the simplex.

Returns

Integer corresponding to the number of simplices removed.

9.14.4.41 size()

```
template<typename traits >
template<std::size_t k>
std::size_t casc::simplicial_complex< traits >::size ( ) const  [inline]
```

Template Parameters

k	The dimension of interest.

Returns

Integer number of k-simplices in the complex.

9.14.4.42 up() [1/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k+1 > > casc::simplicial_complex< traits >::up (
            const SimplexID< k > nid ) const  [inline]
```

Parameters

nid	The simplex of interest

Template Parameters

k	The dimension of the simplex.

Returns

Set of (k+1)-simplices of which 'nid' is a face of.

9.14.4.43 up() [2/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k+1 > > casc::simplicial_complex< traits >::up (
            const std::set< SimplexID< k > > && simplices ) const  [inline]
```

Parameters

simplices	The set of simplices

Template Parameters

k	The dimension of the simplices.

Returns

The set of coboundary simplices.

9.14.4.44 up() [3/3]

```
template<typename traits >
template<std::size_t k>
std::set< SimplexID< k+1 > > casc::simplicial_complex< traits >::up (
            const std::set< SimplexID< k > > & simplices ) const  [inline]
```

Parameters

simplices	The set of simplices

Template Parameters

k	The dimension of the simplices.

Returns

The set of coboundary simplices.

9.14.5 Friends And Related Function Documentation

9.14.5.1 EdgeID

```
template<typename traits >
friend struct EdgeID  [friend]
```

EdgeID is a friend to simplicial_complex

9.14.5.2 SimplexID

```
template<typename traits >
friend struct SimplexID  [friend]
```

SimplexID is a friend of simplicial_complex

The documentation for this class was generated from the following file:

- include/casc/SimplicialComplex.h

9.15 util::type_get< k, T > Struct Template Reference

Helper to get the kth element from a type_holder.

```
#include <util.h>
```

9.15.1 Detailed Description

template<std::size_t k, typename T>
struct util::type_get< k, T >

This is the empty general template which will be later specialized.

Template Parameters

k	Integer index of the type to retrieve
T	A type_holder queue of typenames

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.16 util::type_get< 0, type_holder< Ts... > > Struct Template Reference

Specialization for terminal case.

`#include <util.h>`

Public Types

- using **type** = typename type_holder< Ts... >::head
  The first type of the type_holder.

9.16.1 Detailed Description

template<typename ... Ts>
struct util::type_get< 0, type_holder< Ts... > >

Template Parameters

Ts	Following typenames

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.17 util::type_get< k, type_holder< Ts... > > Struct Template Reference

Specialization to recursively pop types to get the kth type.

`#include <util.h>`

Public Types

- using **type** = typename type_get< k-1, typename type_holder< Ts... >::tail >::type
  Recurse after popping the first type off.

9.17.1 Detailed Description

template<std::size_t k, typename ... Ts>
struct util::type_get< k, type_holder< Ts... > >

Template Parameters

k	Integral constant of the type to get
Ts	List of typenames

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.18 util::type_holder< Ts > Struct Template Reference

Queue based data structure to hold list of types.

`#include <util.h>`

Static Public Attributes

- static const std::size_t **size** = sizeof ... (Ts)

  Length of the list of types.

9.18.1 Detailed Description

template<typename ... Ts>
struct util::type_holder< Ts >

Types in the type_holder can be accessed by accessing the `head` type. Subsequent types are in the `tail`. See also type_get.

Template Parameters

Ts	List of typenames

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.19 util::type_holder< T, Ts... > Struct Template Reference

Partial specialization to allow FIFO access of typenames.

`#include <util.h>`

Public Types

- using **head** = T

  The first type.
- using **tail** = type_holder< Ts... >

  The following types.

Static Public Attributes

- static const std::size_t **size** = 1 + type_holder<Ts...>::size

  Length of the list of types.

9.19.1 Detailed Description

template<typename T, typename ... Ts>
struct util::type_holder< T, Ts... >

Template Parameters

T	The first typename
Ts	The following typenames

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.20 util::type_map< C, V > Struct Template Reference

Map the types in C into `V<T>`.

`#include <util.h>`

Public Types

- using **type** = typename detail::type_map_helper< C, V >::type

  Tuple of `C<V<T1>, V<T2>, V<T3>, ...>`

9.20.1 Detailed Description

template<class C, template< typename > class V>
struct util::type_map< C, V >

Given a container of types `C<T1,T2,T3,...>` and template template type `V<T>`, this function will apply the types in C to `V<T>`. This produces `C<V<T1>, V<T2>, V<T3>, ...>`.

Template Parameters

C	Container of compile time types.
V	Template template class V<T> to map into.

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.21 util::type_swap< TUPLE, HOLDER_FULL > Struct Template Reference

Move a list of types from one container to another.

`#include <util.h>`

9.21.1 Detailed Description

template<template< class ... > class TUPLE, typename HOLDER_FULL>
struct util::type_swap< TUPLE, HOLDER_FULL >

Template Parameters

TUPLE	Empty container
HOLDER_FULL	Full container

The documentation for this struct was generated from the following file:

- include/casc/util.h

9.22 util::type_swap< TUPLE, HOLDER< Ts... > > Struct Template Reference

Move a list of types from one container to another.

`#include <util.h>`

Public Types

- using **type** = TUPLE< Ts... >

  Empty container filled with typenames from full container.

9.22.1 Detailed Description

template<template< class ... > class TUPLE, template< class ... > class HOLDER, typename ... Ts>
struct util::type_swap< TUPLE, HOLDER< Ts... > >

Template Parameters

TUPLE	Empty container
HOLDER	Full container
Ts	Typenames in full container

The documentation for this struct was generated from the following file:

- include/casc/util.h

Chapter 10

File Documentation

10.1 include/casc/CASCFunctions.h File Reference

Contains various functions that operate on simplicial complexes.

```
#include <iostream>
#include <fstream>
#include "SimplicialComplex.h"
#include "CASCTraversals.h"
#include "SimplexSet.h"
#include "stringutil.h"
```

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<typename Complex >
  void casc::getStar (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the star of a SimplexSet.
- template<typename Complex , typename Simplex >
  void casc::getStar (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Gets the star of a simplex.
- template<typename Complex >
  void casc::getClosure (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the closure of a simplex set.
- template<typename Complex , typename Simplex >
  void casc::getClosure (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Compute the closure of a simplex.
- template<typename Complex >
  void casc::getLink (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)

  Gets the link of a SimplexSet.
- template<typename Complex , typename Simplex >
  void casc::getLink (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)

  Gets the link of a simplex.
- template<typename Complex >
  void casc::writeDOT (const std::string &filename, Complex &F)

  Writes out the topology of an ASC into the dot format.

10.2 CASCFunctions.h

Go to the documentation of this file.

```
00001 /*
00002  * ****************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ****************************************************************************
00023  */
00024
00025 /**
00026  * @file CASCFunctions.h
00027  * @brief Contains various functions that operate on simplicial complexes
00028  */
00029
00030 #pragma once
00031
00032 #include <iostream>
00033 #include <fstream>
00034 #include "SimplicialComplex.h"
00035 #include "CASCTraversals.h"
00036 #include "SimplexSet.h"
00037 #include "stringutil.h"
00038
00039 namespace casc
00040 {
00041 /// @cond detail
00042 /// Namespace for templated helpers of convenience functions
00043 namespace func_detail
00044 {
00045
00046 /**
00047  * @brief      Visitor for grabbing simplices in a BFS traversal. See also
00048  *             visit_BFS_up and visit_BFS_down.
00049  *
00050  * @tparam     Complex  Typename of the complex.
00051  */
00052 template <typename Complex>
00053 struct SimplexAggregator
00054 {
00055     /// Alias for a typed SimplexSet
00056     using SimplexSet = typename casc::SimplexSet<Complex>;
00057
00058     /**
00059      * @brief      Constructor of the aggregator.
00060      *
00061      * @param      p     Pointer to SimplexSet to fill.
00062      */
00063     SimplexAggregator(SimplexSet* p) : pLevels(p) {}
00064
00065     /**
00066      * @brief      Overloaded visit function to append different dimensioned
00067      *             SimplexIDs into the SimplexSet.
00068      *
00069      * @param      F          The complex of interest
00070      * @param[in]  s          SimplexID to visit.
00071      *
00072      * @tparam     k          Dimension of the simplex.
00073      *
00074      * @return     True if the traversal should continue.
00075      */
00076     template <std::size_t k>
00077     bool visit(Complex &, typename Complex::template SimplexID<k> s)
00078     {
00079         // If the simplex isn't there, insert it.
00080         if (pLevels->find(s) == pLevels->template end<k>())
00081         {
00082             pLevels->insert(s);
```

```
00083             return true;
00084         }
00085         else
00086         {
00087             // Everything after has been found already
00088             return false;
00089         }
00090     }
00091     private:
00092         SimplexSet* pLevels;
00093 };
00094
00095 /**
00096  * @brief      Helper for computing the star of a set of simplices.
00097  *
00098  * @tparam     Complex  Typename of the simplicial complex.
00099  */
00100 template <typename Complex>
00101 struct StarHelper
00102 {
00103     /**
00104      * @brief      Iterate over the SimplexSet and compute the star.
00105      *
00106      * @param      F     Complex of interest.
00107      * @param      S     SimplexSet of simplices to compute the star of.
00108      * @param      dest  SimplexSet where the star should go.
00109      *
00110      * @tparam     k     Dimension of the current simplex dimension to traverse.
00111      */
00112     template <std::size_t k>
00113     static void apply(Complex                    &F,
00114                       casc::SimplexSet<Complex> &S,
00115                       casc::SimplexSet<Complex> &dest)
00116     {
00117         auto s = casc::get<k>(S);
00118         for (auto simplex : s)
00119         {
00120             visit_BFS_up(SimplexAggregator<Complex>(&dest), F, simplex);
00121         }
00122     }
00123 };
00124
00125 /**
00126  * @brief      Helper for computing the closure of a set of simplices.
00127  *
00128  * @tparam     Complex  Typename of the simplicial complex.
00129  */
00130 template <typename Complex>
00131 struct ClosureHelper
00132 {
00133     /**
00134      * @brief      Iterate over the SimplexSet and compute the closure
00135      *
00136      * @param      F     Complex of interest.
00137      * @param      S     SimplexSet of simplices to compute the closure of.
00138      * @param      dest  SimplexSet where the closure should go.
00139      *
00140      * @tparam     k     Dimension of the current simplex dimension to traverse.
00141      */
00142     template <std::size_t k>
00143     static void apply(Complex                    &F,
00144                       casc::SimplexSet<Complex> &S,
00145                       casc::SimplexSet<Complex> &dest)
00146     {
00147         auto s = casc::get<k>(S);
00148         for (auto simplex : s)
00149         {
00150             visit_BFS_down(SimplexAggregator<Complex>(&dest), F, simplex);
00151         }
00152     }
00153 };
00154
00155 /**
00156  * @brief      Visitor for printing connectivity of the simplicial complex.
00157  *
00158  * @tparam     Complex  Typename of the simplicial complex.
00159  */
00160 template <typename Complex>
00161 struct GraphVisitor
00162 {
00163     /// ostream to write out to.
00164     std::ostream &fout;
00165
00166     /**
00167      * @brief      Constructor
00168      *
00169      * @param      os    Ostream to print to.
```

```
00170      */
00171     GraphVisitor(std::ostream &os) : fout(os) {}
00172
00173     /**
00174      * @brief      Generic visitor prints the simplices and edge connectivity.
00175      *
00176      * @param[in]  F          Complex of interest.
00177      * @param[in]  s          Simplex to visit.
00178      *
00179      * @tparam     level      Dimension of the simplex.
00180      *
00181      * @return     True
00182      */
00183     template <std::size_t level>
00184     bool visit(const Complex &F, typename Complex::template SimplexID<level> s)
00185     {
00186         auto name = to_string(F.get_name(s));
00187
00188         auto covers = F.get_cover(s);
00189         for (auto cover : covers)
00190         {
00191             auto edge = F.get_edge_up(s, cover);
00192             auto nextName = to_string(F.get_name(edge.up()));
00193             if ((*edge).orientation == 1)
00194             {
00195                 fout « "   \"" « name « "\" -> \""
00196                      « nextName « "\"" « std::endl;
00197             }
00198             else
00199             {
00200                 fout « "   \"" « nextName « "\" -> \""
00201                      « name « "\"" «  std::endl;
00202             }
00203         }
00204         return true;
00205     }
00206
00207     /**
00208      * @brief      Explicit specialization for visiting the second to top level
00209      *             simplices.
00210      *
00211      * @param[in]  F          Complex of interest
00212      * @param[in]  s          Simplex to visit.
00213      *
00214      * @return     True;
00215      */
00216     bool visit(const Complex &F, typename Complex::template SimplexID<Complex::topLevel-1> s)
00217     {
00218
00219         auto name   = to_string(F.get_name(s));
00220         auto covers = F.get_cover(s);
00221         for (auto cover : covers)
00222         {
00223             auto edge = F.get_edge_up(s, cover);
00224             auto nextName = to_string(F.get_name(edge.up()));
00225             auto orient   = (*edge.up()).orientation;
00226             if (orient == 1)
00227             {
00228                 nextName = "+ " + nextName;
00229             }
00230             else
00231             {
00232                 nextName = "- " + nextName;
00233             }
00234             if ((*edge).orientation == 1)
00235             {
00236                 fout « "   \"" « name « "\" -> \""
00237                      « nextName « "\"" « std::endl;
00238             }
00239             else
00240             {
00241                 fout « "   \"" « nextName « "\" -> \""
00242                      « name « "\"" « std::endl;
00243             }
00244         }
00245         return true;
00246     }
00247
00248     /**
00249      * @brief      Explicit specialization for visiting the facets of the
00250      *complex.
00251      *
00252      * @param[in]  F          Complex of interest.
00253      * @param[in]  s          Simplex to visit.
00254      */
00255     void visit(const Complex &, typename Complex::template SimplexID<Complex::topLevel>){}
00256 };
```

```
00257
00258 /**
00259  * @brief      Generic template for printing out DOT meta info.
00260  *
00261  * @tparam     Complex  Typename of the complex.
00262  * @tparam     K        Dimension to go through.
00263  */
00264 template <typename Complex, typename K>
00265 struct DotHelper {};
00266
00267 /**
00268  * @brief      Partial specialization for listing names of simplices.
00269  *
00270  * @tparam     Complex  Typename of the complex.
00271  * @tparam     k        Simplex dimension to traverse.
00272  */
00273 template <typename Complex, std::size_t k>
00274 struct DotHelper<Complex, std::integral_constant<std::size_t, k> >
00275 {
00276     /**
00277      * @brief      Print out a list of simplices in a simplex dimension.
00278      *
00279      * @param      fout  Stream to print to.
00280      * @param[in]  F     Complex of interest.
00281      */
00282     static void printlevel(std::ofstream &fout, const Complex &F)
00283     {
00284         auto nodes = F.template get_level_id<k>();
00285         fout « "subgraph cluster_" « k « " {\n"
00286              « "label=\"Level " « k « "\"\n";
00287         for (auto node : nodes)
00288         {
00289             fout « "\"" « to_string(F.get_name(node)) « "\";";
00290         }
00291         fout « "\n}\n";
00292         DotHelper<Complex, std::integral_constant<std::size_t, k+1> >::printlevel(fout, F);
00293     }
00294 };
00295
00296 /**
00297  * @brief      List the names of simplices at the top level
00298  *
00299  * @tparam     Complex  Typename of the complex.
00300  */
00301 template <typename Complex>
00302 struct DotHelper<Complex, std::integral_constant<std::size_t, Complex::topLevel> >
00303 {
00304     /**
00305      * @brief      Print out a list of facets of the complex.
00306      *
00307      * @param      fout  Stream to print to.
00308      * @param[in]  F     Complex of interest.
00309      */
00310     static void printlevel(std::ofstream &fout, const Complex &F)
00311     {
00312         auto nodes = F.template get_level_id<Complex::topLevel>();
00313         fout « "subgraph cluster_" « Complex::topLevel « " {\n"
00314              « "label=\"Level " « Complex::topLevel « "\"\n";
00315         for (auto node : nodes)
00316         {
00317             auto orient = (*node).orientation;
00318             if (orient == 1)
00319             {
00320                 fout « "\"+ ";
00321             }
00322             else
00323             {
00324                 fout « "\"- ";
00325             }
00326             fout « to_string(F.get_name(node)) « "\";";
00327         }
00328         fout « "\n}\n";
00329     }
00330 };
00331 }   // end namespace func_detail
00332 /// @endcond
00333
00334 /**
00335  * @brief      Gets the star of a SimplexSet.
00336  *
00337  * @param[in]  F        Complex of interest.
00338  * @param[in]  S        SimplexSet to compute the star of.
00339  * @param[out] dest     Destination SimplexSet.
00340  *
00341  * @tparam     Complex  Typename of the complex.
00342  */
00343 template <typename Complex>
```

```
00344 void getStar(Complex &F, casc::SimplexSet<Complex> &S,
00345              casc::SimplexSet<Complex> &dest)
00346 {
00347     using SimplexSet = typename casc::SimplexSet<Complex>;
00348     using RevIndex   = typename SimplexSet::cRevIndex;
00349
00350     // Start at the top and work up. We can assume that if we've seen it then
00351     // everything after has been added.
00352     util::int_for_each<std::size_t, RevIndex>(
00353         func_detail::StarHelper<Complex>(), F, S, dest);
00354 }
00355
00356 /**
00357  * @brief      Gets the star of a simplex.
00358  *
00359  * @param[in]  F        Complex of interest.
00360  * @param      s        Simplex to get the star of.
00361  * @param[out] dest     Destination SimplexSet.
00362  *
00363  * @tparam     Complex  Typename of the complex.
00364  * @tparam     Simplex  Typename of the simplex.
00365  */
00366 template <typename Complex, typename Simplex>
00367 void getStar(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
00368 {
00369     visit_BFS_up(func_detail::SimplexAggregator<Complex>(&dest), F, s);
00370 }
00371
00372 /**
00373  * @brief      Gets the closure of a simplex set.
00374  *
00375  * @param[in]  F        Complex of interest.
00376  * @param[in]  S        SimplexSet to compute the closure of.
00377  * @param[out] dest     Destination SimplexSet
00378  *
00379  * @tparam     Complex  Typename of the complex.
00380  */
00381 template <typename Complex>
00382 void getClosure(Complex &F, casc::SimplexSet<Complex> &S,
00383                 casc::SimplexSet<Complex> &dest)
00384 {
00385     using SimplexSet = typename casc::SimplexSet<Complex>;
00386     using LevelIndex = typename SimplexSet::cLevelIndex;
00387     // Start at the bottom and work down.
00388     // We can assume that everything below has been looked at.
00389     util::int_for_each<std::size_t, LevelIndex>(
00390         func_detail::ClosureHelper<Complex>(), F, S, dest);
00391 }
00392
00393 /**
00394  * @brief      Compute the closure of a simplex.
00395  *
00396  * @param[in]  F        Complex of interest.
00397  * @param[in]  s        Simplex of interest.
00398  * @param[out] dest     Destination SimplexSet.
00399  *
00400  * @tparam     Complex  Typename of the complex.
00401  * @tparam     Simplex  Typename of the simplex.
00402  */
00403 template <typename Complex, typename Simplex>
00404 void getClosure(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
00405 {
00406     visit_BFS_down(func_detail::SimplexAggregator<Complex>(&dest), F, s);
00407 }
00408
00409 /**
00410  * @brief      Gets the link of a SimplexSet.
00411  *
00412  * @param[in]  F        Complex of interest.
00413  * @param[in]  S        SimplexSet to get the link of.
00414  * @param[out] dest     Destination SimplexSet.
00415  *
00416  * @tparam     Complex  Typename of the complex.
00417  */
00418 template <typename Complex>
00419 void getLink(Complex &F, casc::SimplexSet<Complex> &S,
00420              casc::SimplexSet<Complex> &dest)
00421 {
00422     using SimplexSet = typename casc::SimplexSet<Complex>;
00423
00424     SimplexSet star;
00425     SimplexSet closure;
00426     SimplexSet closeStar;
00427     SimplexSet starClose;
00428     getStar(F, S, star);
00429     getClosure(F, star, closeStar);
00430
```

```
00431     getClosure(F, S, closure);
00432     getStar(F, closure, starClose);
00433     casc::set_difference(closeStar, starClose, dest);
00434 }
00435
00436 /**
00437  * @brief      Gets the link of a simplex
00438  *
00439  * @param      F        Complex of interest.
00440  * @param      s        Simplex of interest.
00441  * @param      dest     Destination SimplexSet.
00442  *
00443  * @tparam     Complex  Typename of the complex.
00444  * @tparam     Simplex  Typename of the simplex.
00445  */
00446 template <typename Complex, typename Simplex>
00447 void getLink(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
00448 {
00449     using SimplexSet = typename casc::SimplexSet<Complex>;
00450     SimplexSet star;
00451     SimplexSet closure;
00452     SimplexSet closeStar;
00453     SimplexSet starClose;
00454     getStar(F, s, star);
00455     getClosure(F, star, closeStar);
00456
00457     getClosure(F, s, closure);
00458     getStar(F, closure, starClose);
00459     casc::set_difference(closeStar, starClose, dest);
00460 }
00461
00462 /**
00463  * @brief      Writes out the topology of an ASC into the dot format.
00464  *
00465  * The resulting dot file can be rendered into an image using tools such as
00466  * GraphViz.
00467  * ~~~~~~~~~~~~~~~~~~~~~~~{.sh}
00468  * dot -Tpng input.dot > output.png
00469  * ~~~~~~~~~~~~~~~~~~~~~~~
00470  *
00471  * @param[in]  filename  Filename to write out to.
00472  * @param[in]  F         Simplicial complex to generate the DOT of.
00473  *
00474  * @tparam     Complex   Typename of the simplicial complex.
00475  */
00476 template <typename Complex>
00477 void writeDOT(const std::string &filename, Complex &F)
00478 {
00479 // TODO: Put back the const F (0)
00480     std::ofstream fout(filename);
00481     if (!fout.is_open())
00482     {
00483         std::cerr « "File '" « filename
00484                   « "' could not be writen to." « std::endl;
00485         fout.close();
00486         exit(1);
00487     }
00488
00489     fout « "digraph {\n"
00490          « "node [shape = record,height = .1]"
00491          « "splines=line;\n"
00492          « "dpi=300;\n";
00493     auto v = func_detail::GraphVisitor<Complex>(fout);
00494     visit_BFS_up(v, F, F.get_simplex_up());
00495
00496     // List the simplices
00497     func_detail::DotHelper<Complex,
00498                            std::integral_constant<std::size_t, 0> >::printlevel(fout, F);
00499     fout « "}\n";
00500     fout.close();
00501 }
00502 } // end namespace casc
```

10.3 include/casc/CASCTraversals.h File Reference

Implementations of various advanced traversals such as by neighborhood and breadth first search.

```
#include <set>
#include <vector>
```

```
#include <iostream>
#include <string>
#include <type_traits>
#include <utility>
#include <casc/casc>
```

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<typename Visitor , typename SimplexID >
  void casc::visit_BFS_up (Visitor &&v, typename SimplexID::complex &F, SimplexID s)

  Traverse BFS up the complex and apply a visitor function to each simplex visited.
- template<typename Visitor , typename SimplexID >
  void casc::visit_BFS_down (Visitor &&v, typename SimplexID::complex &F, SimplexID s)

  Traverse BFS down the complex and apply a visitor function to each simplex visited.
- template<typename Visitor , typename EdgeID >
  void casc::edge_up (Visitor &&v, typename EdgeID::complex &F, EdgeID s)

  Traverse across edges BFS.
- template<class Complex , std::size_t level, class InsertIter >
  void casc::neighbors (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)

  Push the immediate face neighbors into the provided iterator.
- template<class Complex , class SimplexID , class InsertIter >
  void casc::neighbors (Complex &F, SimplexID nid, InsertIter iter)

  This is a helper function to assist neighbors to automatically deduce the integral level.
- template<class Complex , std::size_t level, class InsertIter >
  void casc::neighbors_up (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)

  Push the immediate coface neighbors into the provided iterator.
- template<class Complex , class SimplexID , class InsertIter >
  void casc::neighbors_up (Complex &F, SimplexID nid, InsertIter iter)

  This is a helper function to assist neighbors to automatically deduce the integral level.
- template<class Complex , std::size_t level, typename Iterator >
  void casc::kneighbors_up (Complex &F, int ring, std::set< typename Complex::template SimplexID< level > > &nbors, Iterator begin, Iterator end)

  Code for returning a set of k-ring neighbors.
- template<class Complex , class SimplexID >
  void casc::kneighbors_up (Complex &F, SimplexID nid, int ring, std::set< SimplexID > &nbors)

  Helper function to help kneighbors_up to deduce the integral level of SimplexID.
- template<class Complex , std::size_t level, typename Iterator >
  void casc::kneighbors (Complex &F, int ring, std::set< typename Complex::template SimplexID< level > > &nbors, Iterator begin, Iterator end)

  Code for returning a set of k-ring neighbors.
- template<class Complex , class SimplexID >
  void casc::kneighbors (Complex &F, SimplexID nid, int ring, std::set< SimplexID > &nbors)

  Helper function to help kneighbors to deduce the integral level of SimplexID.

10.4 CASCTraversals.h

Go to the documentation of this file.

```
00001 /*
00002  * ****************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ****************************************************************************
00023  */
00024
00025 /**
00026  * @file  CASCTraversals.h
00027  * @brief Implementations of various advanced traversals such as by neighborhood
00028  *        and breadth first search.
00029  */
00030
00031 #pragma once
00032
00033 #include <set>
00034 #include <vector>
00035 #include <iostream>
00036 #include <string>
00037 #include <type_traits>
00038 #include <utility>
00039 #include <casc/casc>
00040
00041 namespace casc
00042 {
00043 /// @cond detail
00044 /// Visitor design pattern helper templates
00045 namespace visitor_detail
00046 {
00047
00048 /**
00049  * @brief      General template for BFS up helper.
00050  *
00051  * @tparam     Visitor  Type of visitor functor.
00052  * @tparam     Traits   Traits of the BFS traversal.
00053  * @tparam     Complex  Typename of the simplicial_complex.
00054  * @tparam     K        Current simplex dimension to traverse.
00055  */
00056 template <typename Visitor, typename Traits, typename Complex, typename K>
00057 struct BFS_Up_Node {};
00058
00059 /**
00060  * @brief      Partial specialization for BFS up helper for non facet
00061  *             dimensions.
00062  *
00063  * @tparam     Visitor  Type of visitor functor.
00064  * @tparam     Traits   Traits of the BFS traversal.
00065  * @tparam     Complex  Typename of the simplicial_complex.
00066  * @tparam     k        Current simplex dimension to traverse.
00067  */
00068 template <typename Visitor, typename Traits, typename Complex, std::size_t k>
00069 struct BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
00070 {
00071     /// Simplex dimension currently traversed
00072     static constexpr auto level = k;
00073     /// Typename of the current simplex
00074     using CurrSimplexID = typename Complex::template SimplexID<level>;
00075     /// Typename of coboundary simplices
00076     using NextSimplexID = typename Complex::template SimplexID<level+1>;
00077     /// Container to use to hold coboundary simplices for next recursion.
00078     template <typename T> using Container = typename Traits::template Container<T>;
00079
00080     /// Alias for the recursive call
00081     using BFS_Up_Node_Next = BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t,
      level+1> >;
```

```
00082
00083     /**
00084      * @brief      Visit simplices in the current dimension and continue.
00085      *
00086      * @param[in]  v          Visitor functor.
00087      * @param[in]  F          The simplicial_complex to traverse.
00088      * @param[in]  begin      Iterator to simplices to traverse.
00089      * @param[in]  end        Iterator to end of simplices to traverse.
00090      *
00091      * @tparam     Iterator   Typename of the iterator.
00092      */
00093     template <typename Iterator>
00094     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00095     {
00096         Container<NextSimplexID> next;
00097
00098         for (auto curr = begin; curr != end; ++curr)
00099         {
00100             if (v.visit(F, *curr))
00101             {
00102                 F.get_cover(*curr, [&](typename Complex::KeyType a)
00103                         {
00104                             auto id = F.get_simplex_up(*curr, a);
00105                             next.insert(id);
00106                         });
00107             }
00108         }
00109
00110         BFS_Up_Node_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
00111     }
00112 };
00113
00114 /**
00115  * @brief      Partial specialization for BFS up helper for facets.
00116  *
00117  * @tparam     Visitor  Type of visitor functor.
00118  * @tparam     Traits   Traits of the BFS traversal.
00119  * @tparam     Complex  Typename of the simplicial_complex.
00120  */
00121 template <typename Visitor, typename Traits, typename Complex>
00122 struct BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, Complex::topLevel> >
00123 {
00124     /// Simplex dimension of facets
00125     static constexpr auto level = Complex::topLevel;
00126     /// Typename of the current simplices
00127     using CurrSimplexID = typename Complex::template SimplexID<level>;
00128
00129     /**
00130      * @brief      Visit simplices in the current dimension and continue.
00131      *
00132      * @param[in]  v          Visitor functor.
00133      * @param[in]  F          The simplicial_complex to traverse.
00134      * @param[in]  begin      Iterator to simplices to traverse.
00135      * @param[in]  end        Iterator to end of simplices to traverse.
00136      *
00137      * @tparam     Iterator   Typename of the iterator.
00138      */
00139     template <typename Iterator>
00140     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00141     {
00142         for (auto curr = begin; curr != end; ++curr)
00143         {
00144             v.visit(F, *curr);
00145         }
00146     }
00147 };
00148
00149
00150 /**
00151  * @brief      General template for BFS down helper.
00152  *
00153  * @tparam     Visitor  Type of visitor functor.
00154  * @tparam     Traits   Traits of the BFS traversal.
00155  * @tparam     Complex  Typename of the simplicial_complex.
00156  * @tparam     K        Current simplex dimension to traverse.
00157  */
00158 template <typename Visitor, typename Traits, typename Complex, typename K>
00159 struct BFS_Down_Node {};
00160
00161 /**
00162  * @brief      Partial specialization for BFS down helper for non facet
00163  *             dimensions.
00164  *
00165  * @tparam     Visitor  Type of visitor functor.
00166  * @tparam     Traits   Traits of the BFS traversal.
00167  * @tparam     Complex  Typename of the simplicial_complex.
00168  * @tparam     k        Current simplex dimension to traverse.
```

```
00169  */
00170 template <typename Visitor, typename Traits, typename Complex, std::size_t k>
00171 struct BFS_Down_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
00172 {
00173     /// Simplex dimension current traversed
00174     static constexpr auto level = k;
00175     /// Typename of the current simplices
00176     using CurrSimplexID = typename Complex::template SimplexID<level>;
00177     /// Typename of boundary simplices
00178     using NextSimplexID = typename Complex::template SimplexID<level-1>;
00179     /// Container to use to hold boundary simplices for next recursion
00180     template <typename T> using Container = typename Traits::template Container<T>;
00181
00182     /// Alias for the recursive call
00183     using BFS_Down_Node_Next = BFS_Down_Node<Visitor, Traits, Complex,
       std::integral_constant<std::size_t, level-1> >;
00184
00185     /**
00186      * @brief      Visit simplices in the current dimension and continue.
00187      *
00188      * @param[in]  v          Visitor functor.
00189      * @param[in]  F          The simplicial_complex to traverse.
00190      * @param[in]  begin      Iterator to simplices to traverse.
00191      * @param[in]  end        Iterator to end of simplices to traverse.
00192      *
00193      * @tparam     Iterator   Typename of the iterator.
00194      */
00195     template <typename Iterator>
00196     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00197     {
00198         Container<NextSimplexID> next;
00199
00200         for (auto curr = begin; curr != end; ++curr)
00201         {
00202             if (v.visit(F, *curr))
00203             {
00204                 F.get_name(*curr, [&](typename Complex::KeyType a)
00205                         {
00206                             auto id = F.get_simplex_down(*curr, a);
00207                             next.insert(id);
00208                         });
00209             }
00210         }
00211
00212         BFS_Down_Node_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
00213     }
00214 };
00215
00216 /**
00217  * @brief      Partial specialization for BFS down helper for vertices
00218  *
00219  * @tparam     Visitor  Type of visitor functor.
00220  * @tparam     Traits   Traits of the BFS traversal.
00221  * @tparam     Complex  Typename of the simplicial_complex.
00222  */
00223 template <typename Visitor, typename Traits, typename Complex>
00224 struct BFS_Down_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, 1> >
00225 {
00226     /**
00227      * @brief      Visit simplices in the current dimension and continue.
00228      *
00229      * @param[in]  v          Visitor functor.
00230      * @param[in]  F          The simplicial_complex to traverse.
00231      * @param[in]  begin      Iterator to simplices to traverse.
00232      * @param[in]  end        Iterator to end of simplices to traverse.
00233      *
00234      * @tparam     Iterator   Typename of the iterator.
00235      */
00236     template <typename Iterator>
00237     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00238     {
00239         for (auto curr = begin; curr != end; ++curr)
00240         {
00241             v.visit(F, *curr);
00242         }
00243     }
00244 };
00245
00246 /**
00247  * @deprecated
00248  * @brief      Case to catch accidents... calling down on root is bad.
00249  *
00250  * @tparam     Visitor  { description }
00251  * @tparam     Traits   { description }
00252  * @tparam     Complex  { description }
00253  */
00254 // template <typename Visitor, typename Traits, typename Complex>
```

```
00255 // struct BFS_Down_Node<Visitor, Traits, Complex,
00256 // std::integral_constant<std::size_t,0»
00257 // {
00258 //     template <typename Iterator>
00259 //     static void apply(Visitor&& v, Complex& F, Iterator begin, Iterator end)
00260 //     {}
00261 // };
00262
00263
00264 /**
00265  * @brief      General tempalte for BFS traversal across edges.
00266  *
00267  * @tparam     Visitor  Type of visitor functor.
00268  * @tparam     Traits   Traits of the BFS traversal.
00269  * @tparam     Complex  Typename of the simplicial_complex.
00270  * @tparam     K        Current simplex dimension to traverse.
00271  */
00272 template <typename Visitor, typename Traits, typename Complex, typename K>
00273 struct BFS_Edge {};
00274
00275
00276 /**
00277  * @brief      Partial specialization for BFS Edge for non facets.
00278  *
00279  * @tparam     Visitor  Type of visitor functor.
00280  * @tparam     Traits   Traits of the BFS traversal.
00281  * @tparam     Complex  Typename of the simplicial_complex.
00282  * @tparam     k        Current simplex dimension to traverse.
00283  */
00284 template <typename Visitor, typename Traits, typename Complex, std::size_t k>
00285 struct BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
00286 {
00287     /// Current simplex dimension to traverse
00288     static constexpr auto level = k;
00289     /// Typename of the current EdgeID
00290     using CurrEdgeID = typename Complex::template EdgeID<level>;
00291     /// Typename of the next EdgeID
00292     using NextEdgeID = typename Complex::template EdgeID<level+1>;
00293     /// Typename of the current SimplexID
00294     using CurrSimplexID = typename Complex::template SimplexID<level>;
00295     /// Container to use to hold coboundary edges for next recursion.
00296     template <typename T> using Container = typename Traits::template Container<T>;
00297     /// Alias for the recursive call
00298     using BFS_Edge_Next = BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t,
      level+1> >;
00299
00300     /**
00301      * @brief      Visit simplices in the current dimension and continue.
00302      *
00303      * @param[in]  v          Visitor functor.
00304      * @param[in]  F          The simplicial_complex to traverse.
00305      * @param[in]  begin      Iterator to edges to traverse.
00306      * @param[in]  end        Iterator to end of edges to traverse.
00307      *
00308      * @tparam     Iterator   Typename of the iterator.
00309      */
00310     template <typename Iterator>
00311     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00312     {
00313         Container<NextEdgeID>                  next;
00314         std::vector<typename Complex::KeyType> cover;
00315
00316         for (auto curr = begin; curr != end; ++curr)
00317         {
00318             v.visit(F, *curr);
00319
00320             CurrSimplexID n = curr->up();
00321             F.get_cover(n, std::back_inserter(cover));
00322             for (auto a : cover)
00323             {
00324                 NextEdgeID id = F.get_edge_up(n, a);
00325                 next.insert(next.end(), id);
00326             }
00327             cover.clear();
00328         }
00329
00330         BFS_Edge_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
00331     }
00332 };
00333
00334
00335 /**
00336  * @brief      Partial specialization for BFS Edge for facets.
00337  *
00338  * @tparam     Visitor  Type of visitor functor.
00339  * @tparam     Traits   Traits of the BFS traversal.
00340  * @tparam     Complex  Typename of the simplicial_complex.
```

```
00341  */
00342 template <typename Visitor, typename Traits, typename Complex>
00343 struct BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t, Complex::topLevel> >
00344 {
00345     /// Simplex dimension current traversed
00346     static constexpr auto level = Complex::topLevel;
00347     /// Typename of the current edge
00348     using CurrEdgeID = typename Complex::template EdgeID<level>;
00349
00350     /**
00351      * @brief      Visit the edges to facets.
00352      *
00353      * @param[in]  v          Visitor functor.
00354      * @param[in]  F          The simplicial_complex to traverse.
00355      * @param[in]  begin      Iterator to edges to traverse.
00356      * @param[in]  end        Iterator to end of edges to traverse.
00357      *
00358      * @tparam     Iterator   Typename of the iterator.
00359      */
00360     template <typename Iterator>
00361     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00362     {
00363         for (auto curr = begin; curr != end; ++curr)
00364         {
00365             v.visit(F, *curr);
00366         }
00367     }
00368 };
00369
00370 /// Allow repeat visits of simplices for BFS visits.
00371 struct BFS_Repeat_Node_traits
00372 {
00373     /// Use a vector to allow duplicates
00374     template <typename T> using Container = std::vector<T>;
00375 };
00376
00377 /// No repeat traits for BFS simplex visitor.
00378 struct BFS_NoRepeat_Node_Traits
00379 {
00380     /// Use a NodeSet to avoid duplicates
00381     template <typename T> using Container = NodeSet<T>;
00382 };
00383
00384 /// No repeat traits for BFS edge visitor.
00385 struct BFS_NoRepeat_Edge_Traits
00386 {
00387     /// Use a NodeSet to avoid duplicates.
00388     template <typename T> using Container = NodeSet<T>;
00389     // template <typename Complex, typename SimplexID> auto node_next(Complex F,
00390     // SimplexID s);
00391 };
00392 }   // End namespace visitor_detail
00393 /// @endcond
00394
00395 /**
00396  * @brief      Traverse BFS up the complex and apply a visitor function to each
00397  *             simplex visited.
00398  *
00399  * @param[in]  v          Visitor functor to apply.
00400  * @param      F          The simplicial_complex to traverse.
00401  * @param[in]  s          The simplex to start at.
00402  *
00403  * @tparam     Visitor    Typename of the functor.
00404  * @tparam     SimplexID  Typename of the simplex.
00405  */
00406 template <typename Visitor, typename SimplexID>
00407 void visit_BFS_up(Visitor &&v, typename SimplexID::complex &F, SimplexID s)
00408 {
00409     namespace cvd = visitor_detail;
00410     cvd::BFS_Up_Node<Visitor, cvd::BFS_NoRepeat_Node_Traits, typename SimplexID::complex,
00411                      std::integral_constant<std::size_t, SimplexID::level> >::apply(
00412         std::forward<Visitor>(v), F, &s, &s+1);
00413 }
00414
00415 /**
00416  * @brief      Traverse BFS down the complex and apply a visitor function to
00417  * each
00418  *             simplex visited.
00419  *
00420  * @param[in]  v          Visitor functor to apply.
00421  * @param      F          The simplicial_complex to traverse.
00422  * @param[in]  s          The simplex to start at.
00423  *
00424  * @tparam     Visitor    Typename of the functor.
00425  * @tparam     SimplexID  Typename of the simplex.
00426  */
00427 template <typename Visitor, typename SimplexID>
```

```
00428 void visit_BFS_down(Visitor &&v, typename SimplexID::complex &F, SimplexID s)
00429 {
00430     namespace cvd = visitor_detail;
00431     cvd::BFS_Down_Node<Visitor, cvd::BFS_NoRepeat_Node_Traits, typename SimplexID::complex,
00432                        std::integral_constant<std::size_t, SimplexID::level> >::apply(
00433         std::forward<Visitor>(v), F, &s, &s+1);
00434 }
00435
00436 /**
00437  * @brief      Traverse across edges BFS.
00438  *
00439  * @param[in]  v          Visitor functor to apply.
00440  * @param      F          The simplicial_complex to traverse.
00441  * @param[in]  s          The edge to start at.
00442  *
00443  * @tparam     Visitor    Typename of the functor.
00444  * @tparam     EdgeID     Typename of the edge.
00445  */
00446 template <typename Visitor, typename EdgeID>
00447 void edge_up(Visitor &&v, typename EdgeID::complex &F, EdgeID s)
00448 {
00449     namespace cvd = visitor_detail;
00450     cvd::BFS_Edge<Visitor, cvd::BFS_NoRepeat_Edge_Traits, typename EdgeID::complex,
00451                   std::integral_constant<std::size_t, EdgeID::level> >::apply(
00452         std::forward<Visitor>(v), F, &s, &s+1);
00453 }
00454
00455
00456 /**
00457  * @brief      Push the immediate face neighbors into the provided iterator.
00458  *
00459  *  This function gets the set of neighbors which share a common face. We
00460  *  compute this by traversing to all faces of the simplex of interest. Then we
00461  *  get all cofaces of this set. Depending on the type of iterator passed,
00462  *  duplicate simplices will be included or excluded. Note that this is the
00463  *  traditional definition of neighbor. For example, faces which share an edge
00464  *  are neighbors.
00465  *
00466  * @param      F           The simplicial complex
00467  * @param[in]  nid         Simplex to get neighbors of.
00468  * @param[in]  iter        The iterator to push members into.
00469  *
00470  * @tparam     Complex     Type of the simplicial complex
00471  * @tparam     level       The integral level of the node
00472  * @tparam     InsertIter  Typename of the iterator.
00473  */
00474 template <class Complex, std::size_t level, class InsertIter>
00475 void neighbors(Complex &F, typename Complex::template SimplexID<level> nid, InsertIter iter)
00476 {
00477     for (auto a : F.get_name(nid))
00478     {
00479         auto id = F.get_simplex_down(nid, a);
00480         for (auto b : F.get_cover(id))
00481         {
00482             auto nbor = F.get_simplex_up(id, b);
00483             if (nbor != nid)
00484             {
00485                 *iter++ = nbor;
00486             }
00487         }
00488     }
00489 }
00490
00491 /**
00492  * @brief      This is a helper function to assist neighbors to automatically
00493  *             deduce the integral level.
00494  *
00495  * @param      F           The simplicial complex.
00496  * @param[in]  nid         Simplex to get neighbors of.
00497  * @param[in]  iter        The iterator to push members into.
00498  *
00499  * @tparam     Complex     Type of the simplicial complex
00500  * @tparam     level       The integral level of the node
00501  * @tparam     InsertIter  Typename of the iterator.
00502  */
00503 template <class Complex, class SimplexID, class InsertIter>
00504 void neighbors(Complex &F, SimplexID nid, InsertIter iter)
00505 {
00506     neighbors<Complex, SimplexID::level, InsertIter>(F, nid, iter);
00507 }
00508
00509 /**
00510  * @brief      Push the immediate coface neighbors into the provided iterator.
00511  *
00512  * @param      F           The simplicial complex.
00513  * @param[in]  nid         Simplex to get neighbors of.
00514  * @param[in]  iter        The iterator to push members into.
```

```
00515  *
00516  * @tparam     Complex     Type of the simplicial complex
00517  * @tparam     level       The integral level of the node
00518  * @tparam     InsertIter  Typename of the iterator.
00519  */
00520 template <class Complex, std::size_t level, class InsertIter>
00521 void neighbors_up(Complex &F, typename Complex::template SimplexID<level> nid, InsertIter iter)
00522 {
00523     for (auto a : F.get_cover(nid))
00524     {
00525         auto id = F.get_simplex_up(nid, a);
00526         for (auto b : F.get_name(id))
00527         {
00528             auto nbor = F.get_simplex_down(id, b);
00529             if (nbor != nid)
00530             {
00531                 *iter++ = nbor;
00532             }
00533         }
00534     }
00535 }
00536
00537 /**
00538  * @brief      This is a helper function to assist neighbors to automatically
00539  *             deduce the integral level.
00540  *
00541  * @param      F           The simplicial complex.
00542  * @param[in]  nid         Simplex to get neighbors of.
00543  * @param[in]  iter        The iterator to push members into.
00544  *
00545  * @tparam     Complex     Type of the simplicial complex
00546  * @tparam     level       The integral level of the node
00547  * @tparam     InsertIter  Typename of the iterator.
00548  */
00549 template <class Complex, class SimplexID, class InsertIter>
00550 void neighbors_up(Complex &F, SimplexID nid, InsertIter iter)
00551 {
00552     neighbors_up<Complex, SimplexID::level, InsertIter>(F, nid, iter);
00553 }
00554
00555
00556
00557 /**
00558  * @brief      Code for returning a set of k-ring neighbors.
00559  *
00560  * @param[in]  F         The simplicial_complex to traverse.
00561  * @param[in]  ring      The number of rings of neighbors to collect.
00562  * @param[out] nbors     Set of previously seen simplices.
00563  * @param[in]  begin     The begin
00564  * @param[in]  end       The end
00565  *
00566  * @tparam     Complex   Typename of the simplicial_complex.
00567  * @tparam     level     Simplex dimension of the simplex and neighbors.
00568  * @tparam     Iterator  { description }
00569  */
00570 template <class Complex, std::size_t level, typename Iterator>
00571 void kneighbors_up(Complex                                                  &F,
00572                    int                                                       ring,
00573                    std::set<typename Complex::template SimplexID<level> > &nbors,
00574                    Iterator                                                  begin,
00575                    Iterator                                                  end)
00576 {
00577     if (ring == 0)
00578     {
00579         return;
00580     }
00581     std::set<typename Complex::template SimplexID<level> > next;
00582     for (auto nid = begin; nid != end; ++nid)
00583     {
00584         for (auto a : F.get_cover(*nid))
00585         {
00586             auto id = F.get_simplex_up(*nid, a);
00587             for (auto b : F.get_name(id))
00588             {
00589                 auto nbor = F.get_simplex_down(id, b);
00590                 if (nbors.insert(nbor).second)
00591                 {
00592                     next.insert(nbor);
00593                 }
00594             }
00595         }
00596     }
00597     return kneighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
00598 }
00599
00600 /**
00601  * @brief      Helper function to help kneighbors_up to deduce the integral
```

```
00602  *             level of SimplexID.
00603  *
00604  * @param[in]  F           The simplicial complex
00605  * @param[in]  nid         Simplex of interest to get the nieghbors of.
00606  * @param[in]  ring        The number of rings to include as a neighbor.
00607  * @param[out] nbors       Set of neighbors to populate.
00608  *
00609  * @tparam     Complex     Typename of the complex.
00610  * @tparam     SimplexID   Typename of the SimplexID.
00611  */
00612 template <class Complex, class SimplexID>
00613 void kneighbors_up(Complex &F, SimplexID nid, int ring, std::set<SimplexID> &nbors)
00614 {
00615     nbors.insert(nid);
00616     std::set<SimplexID> next {
00617         nid
00618     };
00619     kneighbors_up<Complex, SimplexID::level>(F, ring, nbors, next.begin(), next.end());
00620     nbors.erase(nid);
00621 }
00622
00623
00624 /**
00625  * @brief      Code for returning a set of k-ring neighbors.
00626  *
00627  * @param[in]  F         The simplicial_complex to traverse.
00628  * @param[in]  ring      The number of rings of neighbors to collect.
00629  * @param[out] nbors     Set of previously seen simplices.
00630  * @param[in]  begin     The begin
00631  * @param[in]  end       The end
00632  *
00633  * @tparam     Complex   Typename of the simplicial_complex.
00634  * @tparam     level     Simplex dimension of the simplex and neighbors.
00635  * @tparam     Iterator  { description }
00636  */
00637 template <class Complex, std::size_t level, typename Iterator>
00638 void kneighbors(Complex                                                  &F,
00639                 int                                                       ring,
00640                 std::set<typename Complex::template SimplexID<level> > &nbors,
00641                 Iterator                                                  begin,
00642                 Iterator                                                  end)
00643 {
00644     if (ring == 0)
00645     {
00646         return;
00647     }
00648     std::set<typename Complex::template SimplexID<level> > next;
00649     for (auto nid = begin; nid != end; ++nid)
00650     {
00651         for (auto a : F.get_name(*nid))
00652         {
00653             auto id = F.get_simplex_down(*nid, a);
00654             for (auto b : F.get_cover(id))
00655             {
00656                 auto nbor = F.get_simplex_up(id, b);
00657                 if (nbors.insert(nbor).second)
00658                 {
00659                     next.insert(nbor);
00660                 }
00661             }
00662         }
00663     }
00664     return kneighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
00665 }
00666
00667 /**
00668  * @brief      Helper function to help kneighbors to deduce the integral
00669  *             level of SimplexID.
00670  *
00671  * @param[in]  F           The simplicial complex
00672  * @param[in]  nid         Simplex of interest to get the nieghbors of.
00673  * @param[in]  ring        The number of rings to include as a neighbor.
00674  * @param[out] nbors       Set of neighbors to populate.
00675  *
00676  * @tparam     Complex     Typename of the complex.
00677  * @tparam     SimplexID   Typename of the SimplexID.
00678  */
00679 template <class Complex, class SimplexID>
00680 void kneighbors(Complex &F, SimplexID nid, int ring, std::set<SimplexID> &nbors)
00681 {
00682     nbors.insert(nid);
00683     std::set<SimplexID> next {
00684         nid
00685     };
00686     kneighbors<Complex, SimplexID::level>(F, ring, nbors, next.begin(), next.end());
00687     nbors.erase(nid);
00688 }
```

```
00689
00690 } // End namespace casc
00691
00692
00693 // namespace visitor_detail
00694 // {
00695 // template <typename Visitor, typename Complex, std::size_t k, std::size_t
00696 // ring>
00697 // struct Neighbors_Up_Node
00698 // {
00699 //     static constexpr auto level = k;
00700 //     using SimplexID = typename Complex::template SimplexID<level>;
00701
00702 //     using Neighbors_Up_Node_Next =
00703 // Neighbors_Up_Node<Visitor,Complex,level,ring-1>;
00704
00705 //     template <typename Iterator>
00706 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00707 // Iterator begin, Iterator end)
00708 //     {
00709 //         NodeSet<SimplexID> next;
00710
00711 //         for(auto curr = begin; curr != end; ++curr)
00712 //         {
00713 //             if(v.visit(F, *curr))
00714 //             {
00715 //                 for(auto a : F.get_cover(*curr))
00716 //                 {
00717 //                     auto id = F.get_simplex_up(*curr,a);
00718 //                     for(auto b : F.get_name(id))
00719 //                     {
00720 //                         auto nbor = F.get_simplex_down(id,b);
00721 //                         if(nodes.insert(nbor).second)
00722 //                         {
00723 //                             next.insert(nbor);
00724 //                         }
00725 //                     }
00726 //                 }
00727 //             }
00728 //         }
00729
00730 //         Neighbors_Up_Node_Next::apply(std::forward<Visitor>(v), F, nodes,
00731 // next.begin(), next.end());
00732 //     }
00733 // };
00734
00735 // template <typename Visitor, typename Complex, std::size_t k>
00736 // struct Neighbors_Up_Node<Visitor, Complex, k, 0>
00737 // {
00738 //     static constexpr auto level = k;
00739 //     using SimplexID = typename Complex::template SimplexID<level>;
00740
00741 //     template <typename Iterator>
00742 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00743 // Iterator begin, Iterator end)
00744 //     {
00745 //         for(auto curr = begin; curr != end; ++curr)
00746 //         {
00747 //             v.visit(F, *curr);
00748 //         }
00749 //     }
00750 // };
00751
00752 // template <typename Visitor, typename Complex, std::size_t k, std::size_t
00753 // ring>
00754 // struct Neighbors_Down_Node
00755 // {
00756 //     static constexpr auto level = k;
00757 //     using SimplexID = typename Complex::template SimplexID<level>;
00758
00759 //     using Neighbors_Down_Node_Next = Neighbors_Down_Node<Visitor,Complex,
00760 //             level,ring-1>;
00761
00762 //     template <typename Iterator>
00763 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00764 // Iterator begin, Iterator end)
00765 //     {
00766 //         NodeSet<SimplexID> next;
00767
00768 //         for(auto curr = begin; curr != end; ++curr)
00769 //         {
00770 //             if(v.visit(F, *curr))
00771 //             {
00772 //                 for(auto a : F.get_name(*curr))
00773 //                 {
00774 //                     auto id = F.get_simplex_down(*curr,a);
00775 //                     for(auto b : F.get_cover(id))
```

```
00776 //                     {
00777 //                         auto nbor = F.get_simplex_up(id,b);
00778 //                         if(nodes.insert(nbor).second)
00779 //                         {
00780 //                             next.insert(nbor);
00781 //                         }
00782 //                     }
00783 //                 }
00784 //             }
00785 //         }
00786
00787 //         Neighbors_Down_Node_Next::apply(std::forward<Visitor>(v), F, nodes,
00788 // next.begin(), next.end());
00789 //     }
00790 // };
00791
00792 // template <typename Visitor, typename Complex, std::size_t k>
00793 // struct Neighbors_Down_Node<Visitor, Complex, k, 0>
00794 // {
00795 //     static constexpr auto level = k;
00796 //     using SimplexID = typename Complex::template SimplexID<level>;
00797
00798 //     template <typename Iterator>
00799 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00800 // Iterator begin, Iterator end)
00801 //     {
00802 //         for(auto curr = begin; curr != end; ++curr)
00803 //         {
00804 //             v.visit(F, *curr);
00805 //         }
00806 //     }
00807 // };
00808 // }
00809
00810
00811 // template <std::size_t rings, typename Visitor, typename SimplexID>
00812 // void visit_neighbors_up(Visitor&& v, typename SimplexID::complex& F,
00813 // SimplexID s)
00814 // {
00815 //     NodeSet<SimplexID> nodes{s};
00816 //     namespace cvd = visitor_detail;
00817 //     cvd::Neighbors_Up_Node<Visitor,typename
00818 // SimplexID::complex,SimplexID::level,rings>::apply(
00819 //             std::forward<Visitor>(v),F,nodes,&s,&s+1);
00820 // }
00821
00822 // template <std::size_t rings, typename Visitor, typename SimplexID>
00823 // void visit_neighbors_down(Visitor&& v, typename SimplexID::complex& F,
00824 // SimplexID s)
00825 // {
00826 //     NodeSet<SimplexID> nodes{s};
00827 //     namespace cvd = visitor_detail;
00828 //     cvd::Neighbors_Down_Node<Visitor,typename
00829 // SimplexID::complex,SimplexID::level,rings>::apply(
00830 //             std::forward<Visitor>(v),F,nodes,&s,&s+1);
00831 // }
```

10.5 include/casc/decimate.h File Reference

Meta-data aware decimation functions.

```
#include <typeinfo>
#include "SimplexSet.h"
#include "SimplexMap.h"
#include "CASCTraversals.h"
#include "CASCFunctions.h"
```

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<typename Complex >
  void casc::perform_removal (Complex &F, casc::SimplexSet< Complex > &S)

  Remove simplex in SimplexSet S from complex F.
- template<typename Complex >
  void casc::perform_insertion (Complex &F, typename decimation_detail::SimplexDataSet< Complex >::type &S)

  Insert all simplices in SimplexSet S into complex F
- template<typename Complex , template< typename > class Callback>
  void casc::run_user_callback (Complex &F, casc::SimplexMap< Complex > &S, Callback< Complex > &&clbk, typename decimation_detail::SimplexDataSet< Complex >::type &rv)

  Run the user specified callback function.
- template<typename Complex , typename Simplex , template< typename > class Callback>
  void casc::decimate (Complex &F, Simplex s, Callback< Complex > &&clbk)

  Decimate a simplex of any dimension while considering any meta-data stores on decimated simplices.
- template<typename Complex , typename Simplex >
  Complex::KeyType casc::decimateFirstHalf (Complex &F, Simplex s, SimplexMap< Complex > &simplex↩Map)

  Given a simplex to decimate generate a pre-post mapping.
- template<typename Complex >
  void casc::decimateBackHalf (Complex &F, SimplexMap< Complex > &simplexMap, typename decimation↩_detail::SimplexDataSet< Complex >::type &rv)

  Given a simplexMap and mapped resulting data execute the decimation.

10.6 decimate.h

Go to the documentation of this file.

```
00001 /*
00002  * ****************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ****************************************************************************
00023  */
00024
00025 /**
00026  * @file  decimate.h
00027  * @brief Meta-data aware decimation functions.
00028  */
00029
00030 #pragma once
00031
00032 #include <typeinfo>
00033
00034 #include "SimplexSet.h"
00035 #include "SimplexMap.h"
00036 #include "CASCTraversals.h"
00037 #include "CASCFunctions.h"
00038
00039 #if __has_cpp_attribute(maybe_unused)
```

```
00040 #define MAYBE_UNUSED [[maybe_unused]]
00041 #else
00042 #define MAYBE_UNUSED
00043 #endif
00044
00045 namespace casc
00046 {
00047 /// @cond detail
00048 /// Namespace for decimation related helpers
00049 namespace decimation_detail
00050 {
00051 /**
00052  * @brief      A multi-vector of simplex, data pairs.
00053  *
00054  * @tparam     Complex  Typename of the simplicial_complex.
00055  */
00056 template <typename Complex>
00057 struct SimplexDataSet
00058 {
00059     /// Typename of vertices
00060     using KeyType = typename Complex::KeyType;
00061
00062     /**
00063      * @brief      Data type makes a pair of array of keys to data type.
00064      *
00065      * @tparam     k     Dimension of simplex.
00066      * @tparam     T     Typename of the data.
00067      */
00068     template <std::size_t k, typename T>
00069     struct DataType
00070     {
00071         /// Pair of array to type
00072         using type = std::pair<std::array<KeyType, k>, T>;
00073     };
00074
00075     /**
00076      * @brief      DataType for simplices with no data.
00077      *
00078      * @tparam     k     Dimension of the simplex.
00079      */
00080     template <std::size_t k>
00081     struct DataType<k, void>
00082     {
00083         /// Array of keys
00084         using type = std::array<KeyType, k>;
00085     };
00086
00087     /// Template to resolve NodeData types for DataType.
00088     template <std::size_t j>
00089     using DataSet = typename DataType<j, typename Complex::template NodeData<j> >::type;
00090     /// Sequence of compile time integers.
00091     using LevelIndex = typename std::make_index_sequence<Complex::numLevels>;
00092     /// Tuple of DataSets corresponding to an integral level.
00093     using SimplexIDLevel = typename util::int_type_map<std::size_t,
00094                                                        std::tuple, LevelIndex, DataSet>::type;
00095     /// Helper vector definition for util.
00096     template <class T> using vector = std::vector<T>;
00097     /// Vector of DataTypes for each integral level.
00098     using type = typename util::type_map<SimplexIDLevel, vector>::type;
00099 };
00100
00101 /**
00102  * @brief      Struct functional to get the complete neighborhood around a
00103  *             simplex.
00104  *
00105  * @tparam     Complex  Type of simplicial complex
00106  */
00107 template <typename Complex>
00108 struct GetCompleteNeighborhood
00109 {
00110     /// Alias for SimplexSet
00111     using SimplexSet = typename casc::SimplexSet<Complex>;
00112
00113     /**
00114      * @brief      Constructor
00115      *
00116      * @param      p     SimplexSet to use to pass results back
00117      */
00118     GetCompleteNeighborhood(SimplexSet* p) : pLevels(p) {}
00119
00120     /**
00121      * @brief      Continue traversing, to the next level
00122      *
00123      * @return     True, continue the BFS
00124      */
00125     template <std::size_t level>
00126     bool visit(Complex &, typename Complex::template SimplexID<level>)
```

```
00127     {
00128         return true;
00129     }
00130
00131     /**
00132      * @brief      Terminal case, go back up (visit_node_up).
00133      *
00134      * @param      F          Simplicial Complex
00135      * @param[in]  s          Simplex of interest
00136      *
00137      * @return     False, stop the BFS traversal
00138      */
00139     bool visit(Complex &F, typename Complex::template SimplexID<1> s)
00140     {
00141         visit_BFS_up(
00142             func_detail::SimplexAggregator<Complex>(pLevels), F, s);
00143         return false;
00144     }
00145
00146     private:
00147         /// Pointer to SimplexSet to store the complete neighborhood.
00148         SimplexSet* pLevels;
00149 };
00150
00151 /**
00152  * @brief      Move found simplices from pLevels to pGrab.
00153  *
00154  * @tparam     Complex  Typename of Simplicial Complex
00155  */
00156 template <typename Complex>
00157 struct GrabVisitor
00158 {
00159     /// Alias for SimplexSet
00160     using SimplexSet = typename casc::SimplexSet<Complex>;
00161
00162     /**
00163      * @brief      Constructor
00164      *
00165      * @param      p     SimplexSet with complete neighborhood.
00166      * @param      grab  SimplexSet to store grabbed simplices.
00167      */
00168     GrabVisitor(SimplexSet* p, SimplexSet* grab) : pLevels(p), pGrab(grab) {}
00169
00170     template <std::size_t level>
00171     bool visit(Complex &, typename Complex::template SimplexID<level> s)
00172     {
00173         if (pLevels->find(s) != pLevels->template end<level>())
00174         {
00175             //std::cout « "GrabVisitor (found): " « s « std::endl;
00176             pLevels->erase(s);
00177             pGrab->insert(s);
00178             return true;
00179         }
00180         else
00181         {
00182             return false;
00183         }
00184     }
00185
00186     private:
00187         /// SimplexSet with the complete neighborhood
00188         SimplexSet* pLevels;
00189         /// SimplexSet with grabbed simplices
00190         SimplexSet* pGrab;
00191 };
00192
00193 template <typename Complex, std::size_t BaseLevel>
00194 struct InnerVisitor
00195 {
00196     using SimplexSet = typename casc::SimplexSet<Complex>;
00197     using SimplexMap = typename casc::SimplexMap<Complex>;
00198     using Simplex = typename Complex::template SimplexID<BaseLevel>;
00199     using KeyType = typename Complex::KeyType;
00200
00201     InnerVisitor(SimplexSet* p, Simplex s, KeyType np, SimplexMap* rv)
00202         : pLevels(p), simplex(s), new_point(np), data(rv) {}
00203
00204     /**
00205      * @brief      Overloaded visit function
00206      *
00207      * @param      F          { parameter_description }
00208      * @param[in]  <unnamed>  { parameter_description }
00209      *
00210      * @tparam     OldLevel   { description }
00211      *
00212      * @return     { description_of_the_return_value }
00213      */
```

```
00214     template <std::size_t OldLevel>
00215     bool visit(Complex &F, typename Complex::template SimplexID<OldLevel> s)
00216     {
00217         constexpr std::size_t NewLevel = OldLevel - BaseLevel + 1;
00218 
00219         if (pLevels->find(s) != pLevels->template end<OldLevel>())
00220         {
00221             //std::cout « "InnerVisitor (found): " « s « std::endl;
00222             std::array<KeyType, OldLevel> old_name = F.get_name(s);
00223             std::array<KeyType, BaseLevel> base_name = F.get_name(simplex);
00224             using NewArrayType = std::array<KeyType, NewLevel>;
00225             NewArrayType new_name;
00226 
00227             std::size_t  i = 0;    // new_name
00228             std::size_t  j = 0;    // old_name
00229             std::size_t  k = 0;    // base_name
00230 
00231             new_name[i++] = new_point;
00232 
00233             // Remove base_name from old_name and append to new_name
00234             while (i < NewLevel)
00235             {
00236                 if (k >= BaseLevel) {
00237                     // append to new_name and increment
00238                     new_name[i++] = old_name[j++];
00239                     continue;
00240                 }
00241                 if (base_name[k] == old_name[j])
00242                 {
00243                     // if equivalent than skip the value
00244                     ++j; ++k;
00245                 }
00246                 else
00247                 {
00248                     // append to new_name and increment
00249                     new_name[i++] = old_name[j++];
00250                 }
00251             }
00252 
00253             SimplexSet grab;
00254             visit_BFS_down(GrabVisitor<Complex>(pLevels, &grab), F, s);
00255 
00256             auto &levelMap = casc::get<NewLevel>(*data);
00257             auto  it = levelMap.find(new_name);
00258             if (it != levelMap.end())
00259             {
00260                 it->second.insert(grab);
00261             }
00262             else
00263             {
00264                 MAYBE_UNUSED auto ret = levelMap.insert(
00265                         std::pair<NewArrayType, SimplexSet>(new_name, grab));
00266                 assert(ret.second);
00267             }
00268         }
00269         return true;
00270     }
00271 
00272     private:
00273         SimplexSet* pLevels;
00274         Simplex     simplex;
00275         KeyType     new_point;
00276         SimplexMap* data;
00277 };
00278 
00279 
00280 template <typename Complex>
00281 struct MainVisitor
00282 {
00283     using SimplexSet = typename casc::SimplexSet<Complex>;
00284     using SimplexMap = typename casc::SimplexMap<Complex>;
00285     using KeyType = typename Complex::KeyType;
00286 
00287     MainVisitor(SimplexSet* p, KeyType np, SimplexMap* rv)
00288         : pLevels(p), new_point(np), data(rv) {}
00289 
00290     template <std::size_t level>
00291     bool visit(Complex &F, typename Complex::template SimplexID<level> s)
00292     {
00293         //std::cout « "MainVisitor: " « s « std::endl;
00294         visit_BFS_up(
00295             InnerVisitor<Complex, level>(
00296                 pLevels, s, new_point, data),
00297             F, s);
00298         return true;
00299     }
00300 
```

```
00301     private:
00302         SimplexSet* pLevels;
00303         KeyType     new_point;
00304         SimplexMap* data;
00305 };
00306
00307 template <typename Complex, template<typename> class Callback>
00308 struct RunCallback
00309 {
00310     using SimplexMap = typename casc::SimplexMap<Complex>;
00311     using SimplexSet = typename casc::SimplexSet<Complex>;
00312     using SimplexDataSet = typename SimplexDataSet<Complex>::type;
00313     using KeyType = typename Complex::KeyType;
00314     template <std::size_t level>
00315     using DataType = typename Complex::template NodeData<level>;
00316
00317
00318     template <std::size_t k, typename ReturnType>
00319     struct PerformCallback
00320     {
00321         static void apply(Complex &F, Callback<Complex> &&clbk,
00322                           SimplexDataSet &rv,
00323                           const std::array<KeyType, k> &new_name,
00324                           const SimplexSet &merged)
00325         {
00326             ReturnType rval = clbk(F, new_name, merged);
00327             std::get<k>(rv).push_back(std::make_pair(new_name, rval));
00328         }
00329     };
00330
00331     template <std::size_t k>
00332     struct PerformCallback<k, void>
00333     {
00334         static void apply(Complex &F, Callback<Complex> &&clbk,
00335                           SimplexDataSet &rv,
00336                           const std::array<KeyType, k> &new_name,
00337                           const SimplexSet &merged)
00338         {
00339             clbk(F, new_name, merged);
00340             std::get<k>(rv).push_back(new_name);
00341         }
00342     };
00343
00344
00345     template <std::size_t k>
00346     static void apply(Complex &F, SimplexMap &S,
00347                       Callback<Complex> &&clbk, SimplexDataSet &rv)
00348     {
00349         auto &levelMap = casc::get<k>(S);
00350         for (auto s : levelMap)
00351         {
00352             PerformCallback<k, DataType<k> >::apply(F, std::forward<Callback<Complex> >(clbk),
00353                                                     rv, s.first, s.second);
00354         }
00355     }
00356 };
00357
00358 template <typename Complex>
00359 struct PerformRemoval
00360 {
00361     template <std::size_t k>
00362     static void apply(Complex &F, casc::SimplexSet<Complex> &S)
00363     {
00364         for (auto curr : casc::get<k>(S))
00365             F.remove(curr);
00366     }
00367 };
00368
00369
00370 template <typename Complex>
00371 struct PerformInsertion {
00372     using KeyType = typename Complex::KeyType;
00373
00374     template <std::size_t k, class T>
00375     static void insert(Complex &F, std::pair<std::array<KeyType, k>, T> P)
00376     {
00377         F.insert(P.first, P.second);
00378     }
00379
00380     template <std::size_t k>
00381     static void insert(Complex &F, std::array<KeyType, k> A)
00382     {
00383         F.insert(A);
00384     }
00385
00386     template <std::size_t k>
00387     static void apply(Complex                                   &F,
```

```
00388                       typename SimplexDataSet<Complex>::type &data)
00389     {
00390         for (auto curr : std::get<k>(data))
00391         {
00392             insert(F, curr);
00393         }
00394     }
00395 };
00396
00397 template <typename Complex>
00398 struct DoomedHelper
00399 {
00400     template <std::size_t k>
00401     static void apply(SimplexSet<Complex> &doomed, SimplexMap<Complex> &simplexMap){
00402         auto s = casc::get<k>(simplexMap);
00403         for (auto map : s){
00404             doomed.insert(map.second);
00405         }
00406     }
00407 };
00408 }   // end namespace decimation_detail
00409 /// @endcond
00410
00411
00412 /**
00413  * @brief      Remove simplex in SimplexSet S from complex F
00414  *
00415  * @param      F        The simplicial_complex to remove from.
00416  * @param      S        SimplexSet of simplices to remove.
00417  *
00418  * @tparam     Complex  Typename of complex
00419  */
00420 template <typename Complex>
00421 void perform_removal(Complex &F, casc::SimplexSet<Complex> &S)
00422 {
00423     using SimplexSet = typename casc::SimplexSet<Complex>;
00424     using LevelIndex = typename SimplexSet::cRevIndex;
00425     util::int_for_each<std::size_t, LevelIndex>(
00426         decimation_detail::PerformRemoval<Complex>(), F, S);
00427 }
00428
00429 /**
00430  * @brief      Insert all simplices in SimplexSet `S` into complex `F`
00431  *
00432  * @param      F        The simplicial_complex to insert into.
00433  * @param      S        SimplexSet of simplices to insert.
00434  *
00435  * @tparam     Complex  Typename of complex
00436  */
00437 template <typename Complex>
00438 void perform_insertion(Complex                                                  &F,
00439                        typename decimation_detail::SimplexDataSet<Complex>::type &S)
00440 {
00441     using SimplexSet = typename casc::SimplexSet<Complex>;
00442     using LevelIndex = typename SimplexSet::cLevelIndex;
00443     util::int_for_each<std::size_t, LevelIndex>(
00444         decimation_detail::PerformInsertion<Complex>(), F, S);
00445 }
00446
00447 /**
00448  * @brief      Run the user specified callback function
00449  *
00450  * @param[in]  F          The simplicial_complex
00451  * @param[in]  S          SimplexMap of
00452  * @param[in]  clbk       User specified callback functor
00453  * @param[out] rv         Multi-vector to place results.
00454  *
00455  * @tparam     Complex    Typename of the simplicial_complex
00456  * @tparam     Callback   Typename of the template template callback functor
00457  */
00458 template <typename Complex, template<typename> class Callback>
00459 void run_user_callback(Complex                                                  &F,
00460                        casc::SimplexMap<Complex>                                &S,
00461                        Callback<Complex>                                       &&clbk,
00462                        typename decimation_detail::SimplexDataSet<Complex>::type &rv)
00463 {
00464     using SimplexMap = typename casc::SimplexMap<Complex>;
00465     using LevelIndex = typename SimplexMap::cLevelIndex;
00466
00467     util::int_for_each<std::size_t, LevelIndex>(
00468         decimation_detail::RunCallback<Complex, Callback>(),
00469         F, S, std::forward<Callback<Complex> >(clbk), rv);
00470 }
00471
00472
00473 /**
00474  * @brief      Decimate a simplex of any dimension while considering any
```

```
00475  *             meta-data stores on decimated simplices.
00476  *
00477  * @param[in]  F         simplicial_complex to operate on.
00478  * @param[in]  s         Simplex to decimate.
00479  * @param[in]  clbk      Callback function to map meta-data
00480  *
00481  * @tparam     Complex   Typename of the simplicial_complex
00482  * @tparam     Simplex   Typename of the simplex
00483  * @tparam     Callback  Typename of the template template callback functor
00484  */
00485 template <typename Complex, typename Simplex, template<typename> class Callback>
00486 void decimate(Complex &F, Simplex s, Callback<Complex> &&clbk)
00487 {
00488     /// Alias for SimplexSet
00489     using SimplexSet = typename casc::SimplexSet<Complex>;
00490     /// Alias for SimplexMap
00491     using SimplexMap = typename casc::SimplexMap<Complex>;
00492
00493     // Create the vertex to replace `s`
00494     typename Complex::KeyType np = F.add_vertex();
00495     SimplexSet nbhd;
00496     SimplexMap simplexMap;
00497
00498     // Get the complete neighborhood
00499     visit_BFS_down(
00500         decimation_detail::GetCompleteNeighborhood<Complex>(&nbhd),
00501         F, s);
00502
00503     SimplexSet doomed = nbhd; // Backup the neighborhood
00504     // Call MainVisitor -> InnerVisitor -> GrabVisitor sequence
00505     visit_BFS_down(
00506         decimation_detail::MainVisitor<Complex>(
00507             &nbhd, np, &simplexMap),
00508         F, s);
00509     // Run the user specified callback
00510     typename decimation_detail::SimplexDataSet<Complex>::type rv;
00511     run_user_callback(F, simplexMap, std::forward<Callback<Complex> >(clbk), rv);
00512     perform_removal(F, doomed); // Remove simplices in the neighborhood
00513     perform_insertion(F, rv);   // Insert new simplices
00514 }
00515
00516 /**
00517  * @brief      Given a simplex to decimate generate a pre-post mapping
00518  *
00519  * @param[in]  F           simplicial_complex to operate on.
00520  * @param[in]  s           Simplex to decimate.
00521  * @param      simplexMap  The simplex map to populate
00522  *
00523  * @tparam     Complex     Typename of the simplicial_complex
00524  * @tparam     Simplex     Typename of the simplex
00525  */
00526 template <typename Complex, typename Simplex>
00527 typename Complex::KeyType decimateFirstHalf(Complex &F, Simplex s, SimplexMap<Complex> &simplexMap)
00528 {
00529     /// Alias for SimplexSet
00530     using SimplexSet = typename casc::SimplexSet<Complex>;
00531
00532     // Create the vertex to replace `s`
00533     typename Complex::KeyType np = F.add_vertex();
00534     SimplexSet nbhd;
00535
00536     // Get the complete neighborhood
00537     visit_BFS_down(
00538         decimation_detail::GetCompleteNeighborhood<Complex>(&nbhd),
00539         F, s);
00540
00541     // Call MainVisitor -> InnerVisitor -> GrabVisitor sequence
00542     visit_BFS_down(
00543         decimation_detail::MainVisitor<Complex>(
00544             &nbhd, np, &simplexMap),
00545         F, s);
00546     return np;
00547 }
00548
00549 /**
00550  * @brief      Given a simplexMap and mapped resulting data execute the
00551  *             decimation.
00552  *
00553  * @param      F           Simplicial complex to operate on
00554  * @param      simplexMap  SimplexMap mapping simplices before and after decimation
00555  * @param      rv          Resulting data for each simplex
00556  *
00557  * @tparam     Complex     Typename of the complex of interest
00558  */
00559 template <typename Complex>
00560 void decimateBackHalf(Complex &F, SimplexMap<Complex> &simplexMap, typename
      decimation_detail::SimplexDataSet<Complex>::type &rv){
```

```
00561
00562     SimplexSet<Complex> doomed;
00563     util::int_for_each<std::size_t, typename
      SimplexMap<Complex>::cLevelIndex>(decimation_detail::DoomedHelper<Complex>(), doomed, simplexMap);
00564
00565     perform_removal(F, doomed); // Remove simplices in the neighborhood
00566     perform_insertion(F, rv);   // Insert new simplices
00567 }
00568
00569 } // end namespace casc
```

10.7 include/casc/index_tracker.h File Reference

B-tree based interval tracker.

```
#include <iostream>
#include <assert.h>
#include <array>
#include <vector>
#include <cstdlib>
#include <limits>
```

Data Structures

- struct index_tracker::index_tracker_detail::Interval< T >

  Interval object represents a range.
- struct index_tracker::index_tracker_detail::BTreeNode< _T, _d >

  An array based BTree.
- class index_tracker::index_tracker< _T, _d >

  Tracker of available indices implemented as a B-tree of intervals.

Namespaces

- namespace index_tracker

  Index tracker namespace.
- namespace index_tracker::index_tracker_detail

  B-tree internal data structures.

Typedefs

- template<typename Node >
  using **index_tracker::index_tracker_detail::Pointer** = typename Node::Pointer
- template<typename Node >
  using **index_tracker::index_tracker_detail::Data** = typename Node::Data
- template<typename Node >
  using **index_tracker::index_tracker_detail::Scalar** = typename Node::Scalar

Functions

- template<typename T >
  bool **index_tracker::index_tracker_detail::operator<** (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator>** (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator<** (T x, const Interval< T > &y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator>** (const Interval< T > &x, T y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator<** (const Interval< T > &x, T y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator>** (T x, const Interval< T > &y)
- template<typename T >
  bool **index_tracker::index_tracker_detail::operator==** (const Interval< T > &x, const Interval< T > &y)
- template<typename T >
  std::ostream & **index_tracker::index_tracker_detail::operator<<** (std::ostream &out, const Interval< T > &x)
- template<typename T >
  int **index_tracker::index_tracker_detail::merge** (Interval< T > &A, T x)
- template<typename Node >
  void **index_tracker::index_tracker_detail::rebalance** (Pointer< Node > head, std::size_t i)
- template<typename Node >
  void **index_tracker::index_tracker_detail::insert_H** (Pointer< Node > head, const Data< Node > &data)
- template<typename Node >
  Pointer< Node > **index_tracker::index_tracker_detail::insert** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  bool **index_tracker::index_tracker_detail::get** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  void **index_tracker::index_tracker_detail::get_replacement** (Pointer< Node > head, Data< Node > &key)
- template<typename Node >
  void **index_tracker::index_tracker_detail::remove_H** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  Pointer< Node > **index_tracker::index_tracker_detail::remove** (Pointer< Node > head, Data< Node > data)
- template<typename Node >
  void **index_tracker::index_tracker_detail::fill_left** (Pointer< Node > head, Data< Node > &x)
- template<typename Node >
  void **index_tracker::index_tracker_detail::fill_right** (Pointer< Node > head, Data< Node > &x)
- template<typename Node >
  void **index_tracker::index_tracker_detail::insert_scalar_H** (Pointer< Node > head, Scalar< Node > data)
- template<typename Node >
  Pointer< Node > **index_tracker::index_tracker_detail::insert_scalar** (Pointer< Node > head, Scalar< Node > data)
- template<typename Node >
  void **index_tracker::index_tracker_detail::insert_left** (Pointer< Node > head, const Data< Node > &x)
- template<typename Node >
  bool **index_tracker::index_tracker_detail::remove_scalar_H** (Pointer< Node > head, Scalar< Node > x)
- template<typename Node >
  bool **index_tracker::index_tracker_detail::remove_scalar** (Pointer< Node > &head, Scalar< Node > data)
- template<typename Node >
  Scalar< Node > **index_tracker::index_tracker_detail::pop_scalar** (Pointer< Node > &head)

- template<typename Node >
  void **index_tracker::index_tracker_detail::destruct** (Pointer< Node > head)
- template<typename Node >
  Data< Node > **index_tracker::index_tracker_detail::check_order** (Pointer< Node > head, Data< Node > curr)
- template<typename T , std::size_t d>
  std::ostream & **index_tracker::operator**<< (std::ostream &out, const index_tracker_detail::BTreeNode< T, d > *head)

10.8 index_tracker.h

Go to the documentation of this file.

```
00001 /*
00002  * ****************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ****************************************************************************
00023  */
00024
00025 /**
00026  * @file  index_tracker.h
00027  * @brief B-tree based interval tracker.
00028  */
00029
00030 #pragma once
00031
00032 #include <iostream>
00033 #include <assert.h>
00034 #include <array>
00035 #include <vector>
00036 #include <cstdlib>
00037 #include <limits>
00038
00039
00040 /// Index tracker namespace
00041 namespace index_tracker
00042 {
00043 /// B-tree internal data structures
00044 namespace index_tracker_detail {
00045
00046
00047     /**
00048      * @brief      Interval object represents a range.
00049      *
00050      * @tparam     T     Typename of the interval data
00051      */
00052     template <typename T>
00053     struct Interval
00054     {
00055         /// Default constructor
00056         Interval() : _a(0), _b(0) {}
00057         /// Construct an interval from a to a+1
00058         Interval(T a) : _a(a), _b(a+1) {}
00059         /// Construct an interval from a to b
00060         Interval(T a, T b) : _a(a), _b(b) { assert(a <= b); }
00061         /// Copy constructor
00062         Interval(const Interval<T>& rhs) : _a(rhs._a), _b(rhs._b) {}
00063
00064         /**
00065          * @brief      Assigment operator overload.
```

```
00066          *
00067          * @param[in]  rhs   The right hand side
00068          *
00069          * @return     Reference to this
00070          */
00071         Interval& operator=(const Interval& rhs)
00072         {
00073             _a = rhs._a;
00074             _b = rhs._b;
00075             return *this;
00076         }
00077
00078         /// Is x in the bounds of the interval
00079         bool has(T x) { return _a <= x && x < _b; }
00080
00081         /// Get the lower inclusive bound of the interval
00082         T  lower() const { return _a; }
00083         /// Get the upper exclusive bound of the interval
00084         T  upper() const { return _b; }
00085
00086         /// Get the lower inclusive bound of the interval
00087         T& lower()       { return _a; }
00088         /// Get the upper exclusive bound of the interval
00089         T& upper()       { return _b; }
00090
00091         /// Get the size of the interval
00092         std::size_t size() { return _b - _a; }
00093
00094     private:
00095         T _a;   /// Inclusive lower bound
00096         T _b;   /// Exclusive upper bound
00097     };
00098
00099     template <typename T>
00100     bool operator<(const Interval<T>& x, const Interval<T>& y)
00101     {
00102         return x.upper() <= y.lower();
00103     }
00104
00105     template <typename T>
00106     bool operator>(const Interval<T>& x, const Interval<T>& y)
00107     {
00108         return x.lower() >= y.upper();
00109     }
00110
00111     template <typename T>
00112     bool operator<(T x, const Interval<T>& y)
00113     {
00114         return x < y.lower();
00115     }
00116
00117     template <typename T>
00118     bool operator>(const Interval<T>& x, T y)
00119     {
00120         return x.lower() > y;
00121     }
00122
00123     template <typename T>
00124     bool operator<(const Interval<T>& x, T y)
00125     {
00126         return x.upper() <= y;
00127     }
00128
00129     template <typename T>
00130     bool operator>(T x, const Interval<T>& y)
00131     {
00132         return x >= y.upper();
00133     }
00134
00135     template <typename T>
00136     bool operator==(const Interval<T>& x, const Interval<T>& y)
00137     {
00138         return (x.lower() == y.lower()) && (x.upper() && y.upper());
00139     }
00140
00141     template <typename T>
00142     std::ostream& operator«(std::ostream& out, const Interval<T>& x)
00143     {
00144         out « "[" « x.lower() « "~" « x.upper() « ")";
00145         return out;
00146     }
00147
00148     template <typename T>
00149     int merge(Interval<T>& A, T x)
00150     {
00151         // If x isn't the next lower value return 0
00152         if(x + 1 < A.lower())
```

```
00153             return 0;
00154         else if(x + 1 == A.lower())
00155         {
00156             // if x is the next lowest value assign to lower
00157             A.lower() = x;
00158             return 1;
00159         }
00160         else if(A.lower() <= x && x < A.upper()) // x is in range already
00161             return 2;
00162         else if(A.upper() == x)
00163         {
00164             // x is the next higher assign upper
00165             A.upper() = x + 1;
00166             return 3;
00167         }
00168         else if(A.upper() < x)
00169             // x isn't next after this range return 4
00170             return 4;
00171         else
00172             return 5; // Something undefined happened.
00173     }
00174
00175
00176     /**
00177      * @brief      An array based BTree
00178      *
00179      * @tparam     _T    { description }
00180      * @tparam     _d    { description }
00181      */
00182     template <typename _T, std::size_t _d>
00183     struct BTreeNode
00184     {
00185         static constexpr std::size_t d = _d;
00186         static constexpr std::size_t N = 2*d+1;
00187         using Scalar = _T;
00188         using Data = Interval<Scalar>;
00189         using Pointer = BTreeNode*;
00190
00191         BTreeNode() {}
00192         BTreeNode(const Data& t)
00193             : k(1), next{nullptr,nullptr}
00194         {
00195             data[0] = t;
00196         }
00197
00198         template <typename Iter>
00199         BTreeNode(Iter begin, Iter end)
00200         {
00201             k = 0;
00202             while(begin != end)
00203             {
00204                 next[k] = nullptr;
00205                 data[k++] = *begin++;
00206             }
00207             next[k] = nullptr;
00208         }
00209
00210         std::size_t             k;
00211         std::array<Data,N>      data;
00212         std::array<Pointer,N+1> next;
00213     };
00214
00215     template <typename Node> using Pointer = typename Node::Pointer;
00216     template <typename Node> using Data = typename Node::Data;
00217     template <typename Node> using Scalar = typename Node::Scalar;
00218
00219
00220
00221     template <typename Node>
00222     void rebalance(Pointer<Node> head, std::size_t i)
00223     {
00224         Pointer<Node> curr = head->next[i];
00225
00226         if(curr->k == Node::N)
00227         {
00228             // Pointer<Node> left = curr; // UNUSED
00229             Pointer<Node> right = new Node(curr->data.begin() + Node::d + 1, curr->data.end());
00230             curr->k = Node::d;
00231
00232             if(curr->next[0] == nullptr)
00233             {
00234                 right->next[0] = nullptr;
00235             }
00236             else
00237             {
00238                 for(std::size_t i = 0; i <= Node::d; ++i)
00239                 {
```

```
00240                     right->next[i] = curr->next[Node::d + i + 1];
00241                 }
00242             }
00243
00244             Data<Node> up = curr->data[Node::d];
00245
00246             for(std::size_t j = head->k; j > i; --j)
00247             {
00248                 head->data[j] = head->data[j-1];
00249                 head->next[j+1] = head->next[j];
00250             }
00251             head->data[i] = up;
00252             head->next[i+1] = right;
00253             ++(head->k);
00254         }
00255         else if(curr->k < Node::d)
00256         {
00257             if(i > 0 && head->next[i-1]->k > Node::d)
00258             {
00259                 Pointer<Node> left  = head->next[i-1];
00260                 Pointer<Node> right = head->next[i];
00261
00262                 if(right->next[0] != nullptr)
00263                     right->next[right->k + 1] = right->next[right->k];
00264                 for(std::size_t j = right->k; j > 0; --j)
00265                 {
00266                     right->data[j] = right->data[j-1];
00267                     if(left->next[0] != nullptr)
00268                         right->next[j] = right->next[j-1];
00269                 }
00270                 right->data[0] = head->data[i-1];
00271                 if(left->next[0] != nullptr)
00272                     right->next[0] = left->next[left->k];
00273                 ++(right->k);
00274
00275                 head->data[i-1] = left->data[left->k-1];
00276
00277                 --(left->k);
00278
00279     //          std::cout << "Rotate Right" << std::endl;
00280             }
00281             else if(i < head->k && head->next[i+1]->k > Node::d)
00282             {
00283                 Pointer<Node> left  = head->next[i];
00284                 Pointer<Node> right = head->next[i+1];
00285
00286                 left->data[left->k] = head->data[i];
00287                 ++(left->k);
00288                 if(left->next[0] != nullptr)
00289                     left->next[left->k] = right->next[0];
00290
00291                 head->data[i] = right->data[0];
00292                 for(std::size_t j = 0; j < right->k - 1; ++j)
00293                 {
00294                     right->data[j] = right->data[j+1];
00295                     if(right->next[0] != nullptr)
00296                         right->next[j] = right->next[j+1];
00297                 }
00298                 --(right->k);
00299                 if(right->next[0] != nullptr)
00300                     right->next[right->k] = right->next[right->k + 1];
00301
00302     //          std::cout << "Rotate Left" << std::endl;
00303             }
00304             else
00305             {
00306                 if(i < head->k)
00307                 {
00308                     Pointer<Node> left = head->next[i];
00309                     Pointer<Node> right = head->next[i+1];
00310
00311                     left->data[(left->k)++] = head->data[i];
00312                     for(std::size_t j = 0; j < right->k; ++j)
00313                     {
00314                         left->data[left->k] = right->data[j];
00315                         if(left->next[0] != nullptr)
00316                             left->next[left->k] = right->next[j];
00317                         ++(left->k);
00318                     }
00319                     if(left->next[0] != nullptr)
00320                         left->next[left->k] = right->next[right->k];
00321
00322                     delete right;
00323
00324                     --(head->k);
00325                     for(std::size_t j = i; j < head->k; ++j)
00326                     {
```

```
00327                         head->data[j] = head->data[j+1];
00328                         head->next[j+1] = head->next[j+2];
00329                     }
00330                 }
00331                 else
00332                 {
00333                     Pointer<Node> left = head->next[i-1];
00334                     Pointer<Node> right = head->next[i];
00335 
00336                     left->data[left->k] = head->data[i-1];
00337                     ++(left->k);
00338                     for(std::size_t j = 0; j < right->k; ++j)
00339                     {
00340                         left->data[left->k] = right->data[j];
00341                         if(left->next[0] != nullptr)
00342                             left->next[left->k] = right->next[j];
00343                         ++(left->k);
00344                     }
00345                     if(left->next[0] != nullptr)
00346                         left->next[left->k] = right->next[right->k];
00347 
00348                     delete right;
00349                     --(head->k);
00350                 }
00351             }
00352         }
00353     }
00354 
00355     template <typename Node>
00356     void insert_H(Pointer<Node> head, const Data<Node>& data)
00357     {
00358         if(head->next[0] == nullptr)
00359         {
00360             const auto k = head->k;
00361 
00362             std::size_t i = 0;
00363             while(i < k && head->data[i] < data)
00364             {
00365                 ++i;
00366             }
00367             for(std::size_t j = k; j > i; --j)
00368             {
00369                 head->data[j] = head->data[j-1];
00370             }
00371             head->data[i] = data;
00372             head->k = k+1;
00373         }
00374         else
00375         {
00376             const auto k = head->k;
00377 
00378             std::size_t i = 0;
00379             while(i < k && head->data[i] < data)
00380                 ++i;
00381 
00382             insert_H<Node>(head->next[i], data);
00383             rebalance<Node>(head, i);
00384         }
00385     }
00386 
00387     template <typename Node>
00388     Pointer<Node> insert(Pointer<Node> head, Data<Node> data)
00389     {
00390         if(head == nullptr)
00391         {
00392             return new Node(data);
00393         }
00394         else
00395         {
00396             insert_H<Node>(head, data);
00397             if(head->k == Node::N)
00398             {
00399                 Pointer<Node> nn = new Node();
00400                 nn->k = 0;
00401                 nn->next[0] = head;
00402                 rebalance<Node>(nn, 0);
00403                 return nn;
00404             }
00405             else
00406             {
00407                 return head;
00408             }
00409         }
00410     }
00411 
00412     template <typename Node>
00413     bool get(Pointer<Node> head, Data<Node> data)
```

```
00414     {
00415         if(head->next[0] == nullptr)
00416         {
00417             for(std::size_t i = 0; i < head->k; ++i)
00418             {
00419                 if(data == head->data[i])
00420                 {
00421                     return true;
00422                 }
00423             }
00424             return false;
00425         }
00426         else
00427         {
00428             for(std::size_t i = 0; i < head->k; ++i)
00429             {
00430                 if(data < head->data[i])
00431                 {
00432                     return get<Node>(head->next[i], data);
00433                 }
00434                 else if(data == head->data[i])
00435                 {
00436                     return true;
00437                 }
00438             }
00439             return get<Node>(head->next[head->k], data);
00440         }
00441     }
00442
00443
00444
00445     template <typename Node>
00446     void get_replacement(Pointer<Node> head, Data<Node>& key)
00447     {
00448         if(head->next[0] == nullptr)
00449         {
00450             key = head->data[head->k-1];
00451             --(head->k);
00452         }
00453         else
00454         {
00455             get_replacement<Node>(head->next[head->k], key);
00456             rebalance<Node>(head, head->k);
00457         }
00458     }
00459
00460     template <typename Node>
00461     void remove_H(Pointer<Node> head, Data<Node> data)
00462     {
00463         if(head->next[0] == nullptr)
00464         {
00465             for(std::size_t i = 0; i < head->k; ++i)
00466             {
00467                 if(data == head->data[i])
00468                 {
00469                     for(std::size_t j = i+1; j < head->k; ++j)
00470                     {
00471                         head->data[j-1] = head->data[j];
00472                     }
00473                     --(head->k);
00474                     break;
00475                 }
00476             }
00477         }
00478         else
00479         {
00480             for(std::size_t i = 0; i < head->k; ++i)
00481             {
00482                 if(data < head->data[i])
00483                 {
00484                     remove_H<Node>(head->next[i], data);
00485                     rebalance<Node>(head, i);
00486                     return;
00487                 }
00488                 else if(data == head->data[i])
00489                 {
00490                     get_replacement<Node>(head->next[i], head->data[i]);
00491                     rebalance<Node>(head, i);
00492                     return;
00493                 }
00494             }
00495             remove_H<Node>(head->next[head->k], data);
00496             rebalance<Node>(head, head->k);
00497         }
00498     }
00499
00500     template <typename Node>
```

```
00501     Pointer<Node> remove(Pointer<Node> head, Data<Node> data)
00502     {
00503         remove_H<Node>(head, data);
00504 
00505         if(head->k == 0)
00506         {
00507             Pointer<Node> rval = head->next[0];
00508             delete head;
00509             return rval;
00510         }
00511         else
00512         {
00513             return head;
00514         }
00515     }
00516 
00517 
00518     template <typename Node>
00519     void fill_left(Pointer<Node> head, Data<Node>& x)
00520     {
00521         if(head->next[0] == nullptr)
00522         {
00523             Data<Node>& left = head->data[head->k-1];
00524             if(left.upper() == x.lower())
00525             {
00526                 x.lower() = left.lower();
00527                 --(head->k);
00528             }
00529         }
00530         else
00531         {
00532             fill_left<Node>(head->next[head->k], x);
00533             rebalance<Node>(head, head->k);
00534         }
00535     }
00536 
00537 
00538     template <typename Node>
00539     void fill_right(Pointer<Node> head, Data<Node>& x)
00540     {
00541         if(head->next[0] == nullptr)
00542         {
00543             Data<Node>& right = head->data[0];
00544             if(right.lower() == x.upper())
00545             {
00546                 x.upper() = right.upper();
00547                 --(head->k);
00548                 for(std::size_t i = 0; i < head->k; ++i)
00549                 {
00550                     head->data[i] = head->data[i+1];
00551                 }
00552             }
00553         }
00554         else
00555         {
00556             fill_right<Node>(head->next[0], x);
00557             rebalance<Node>(head, 0);
00558         }
00559     }
00560 
00561 
00562     template <typename Node>
00563     void insert_scalar_H(Pointer<Node> head, Scalar<Node> data)
00564     {
00565         // If the
00566         if(head->next[0] == nullptr)
00567         {
00568             const auto k = head->k;
00569 
00570             std::size_t i;
00571             for(i = 0; i < k; ++i)
00572             {
00573                 Data<Node>& A = head->data[i];
00574                 Scalar<Node> x = data;
00575 
00576                 if(x + 1 < A.lower())
00577                 {
00578                     for(std::size_t j = k; j > i; --j)
00579                     {
00580                         head->data[j] = head->data[j-1];
00581                     }
00582                     head->data[i] = data;
00583                     ++(head->k);
00584                     return;
00585                 }
00586                 else if(x + 1 == A.lower())
00587                 {
```

```
00588                     A.lower() = x;
00589                     return;
00590                 }
00591                 else if(A.lower() <= x && x < A.upper())
00592                 {
00593                     return;
00594                 }
00595                 else if(A.upper() == x)
00596                 {
00597                     if(i + 1 < k)
00598                     {
00599                         Data<Node>& B = head->data[i+1];
00600                         if(x + 1 == B.lower())
00601                         {
00602                             A.upper() = B.upper();
00603                             for(std::size_t j = i+1; j < k-1; ++j)
00604                             {
00605                                 head->data[j] = head->data[j+1];
00606                             }
00607                             --(head->k);
00608                         }
00609                         else
00610                         {
00611                             A.upper() = x + 1;
00612                         }
00613                     }
00614                     else
00615                     {
00616                         A.upper() = x + 1;
00617                     }
00618                     return;
00619                 }
00620             }
00621             head->data[i] = data;
00622             ++(head->k);
00623         }
00624         else
00625         {
00626             const auto k = head->k;
00627
00628             std::size_t i;
00629             for(i = 0; i < k; ++i)
00630             {
00631                 Data<Node>& A = head->data[i];
00632                 Scalar<Node> x = data;
00633
00634                 if(x + 1 < A.lower())
00635                 {
00636                     insert_scalar_H<Node>(head->next[i], data);
00637                     rebalance<Node>(head, i);
00638                     return;
00639                 }
00640                 else if(x + 1 == A.lower())
00641                 {
00642                     A.lower() = x;
00643                     fill_left<Node>(head->next[i], A);
00644                     rebalance<Node>(head, i);
00645                     return;
00646                 }
00647                 else if(A.lower() <= x && x < A.upper())
00648                 {
00649                     return;
00650                 }
00651                 else if(A.upper() == x)
00652                 {
00653                     A.upper() = x + 1;
00654                     fill_right<Node>(head->next[i+1], A);
00655                     rebalance<Node>(head, i+1);
00656                     return;
00657                 }
00658             }
00659             insert_scalar_H<Node>(head->next[i], data);
00660             rebalance<Node>(head, i);
00661         }
00662     }
00663
00664     template <typename Node>
00665     Pointer<Node> insert_scalar(Pointer<Node> head, Scalar<Node> data)
00666     {
00667         if(head == nullptr)
00668         {
00669             return new Node(data);
00670         }
00671         else
00672         {
00673             insert_scalar_H<Node>(head, data);
00674             if(head->k == Node::N)
```

```
00675             {
00676                 Pointer<Node> nn = new Node();
00677                 nn->k = 0;
00678                 nn->next[0] = head;
00679                 rebalance<Node>(nn, 0);
00680                 return nn;
00681             }
00682             else if(head->k == 0)
00683             {
00684                 Pointer<Node> rval = head->next[0];
00685                 delete head;
00686                 return rval;
00687             }
00688             else
00689             {
00690                 return head;
00691             }
00692         }
00693     }
00694
00695     template <typename Node>
00696     void insert_left(Pointer<Node> head, const Data<Node>& x)
00697     {
00698         if(head->next[0] == nullptr)
00699         {
00700             head->data[head->k] = x;
00701             ++(head->k);
00702         }
00703         else
00704         {
00705             insert_left<Node>(head->next[head->k], x);
00706             rebalance<Node>(head, head->k);
00707         }
00708     }
00709
00710
00711     template <typename Node>
00712     bool remove_scalar_H(Pointer<Node> head, Scalar<Node> x)
00713     {
00714         if(head->next[0] == nullptr)
00715         {
00716             const auto k = head->k;
00717
00718             std::size_t i;
00719             for(i = 0; i < k; ++i)
00720             {
00721                 Data<Node>& A = head->data[i];
00722
00723                 if(x < A.lower())
00724                 {
00725     //              std::cout « "if(x < A.lower())" « std::endl;
00726                     return false;
00727                 }
00728                 else if(x == A.lower())
00729                 {
00730     //              std::cout « "if(x == A.lower())" « std::endl;
00731                     if(x + 1 == A.upper())
00732                     {
00733     //                  std::cout « "if(x + 1 == A.upper())" « std::endl;
00734                         --(head->k);
00735                         for(std::size_t j = i; j < head->k; ++j)
00736                         {
00737                             head->data[j] = head->data[j+1];
00738                         }
00739                         return true;
00740                     }
00741                     A.lower() = x + 1;
00742                     return true;
00743                 }
00744                 else if(/*A.lower() < x &&*/ x + 1 < A.upper())
00745                 {
00746     //              std::cout « "x + 1 < A.upper()" « std::endl;
00747                     for(std::size_t j = head->k; j > i; --j)
00748                     {
00749                         head->data[j] = head->data[j-1];
00750                     }
00751                     ++(head->k);
00752                     A.upper() = x;
00753                     head->data[i+1].lower() = x + 1;
00754                     return true;
00755                 }
00756                 else if(x + 1 == A.upper())
00757                 {
00758     //              std::cout « "x + 1 < A.upper()" « std::endl;
00759                     A.upper() = x;
00760                     return true;
00761                 }
```

```
00762             }
00763             return false;
00764         }
00765         else
00766         {
00767             const auto k = head->k;
00768 
00769             std::size_t i;
00770             for(i = 0; i < k; ++i)
00771             {
00772                 Data<Node>& A = head->data[i];
00773 
00774                 if(x < A.lower())
00775                 {
00776                     bool rval = remove_scalar_H<Node>(head->next[i], x);
00777                     rebalance<Node>(head, i);
00778                     return rval;
00779                 }
00780                 else if(x == A.lower())
00781                 {
00782                     if(x + 1 == A.upper())
00783                     {
00784                         get_replacement<Node>(head->next[i], A);
00785                         rebalance<Node>(head, i);
00786                         return true;
00787                     }
00788                     A.lower() = x + 1;
00789                     return true;
00790                 }
00791                 else if(/*A.lower() < x &&*/ x + 1 < A.upper())
00792                 {
00793                     Data<Node> B(A.lower(), x);
00794                     A.lower() = x + 1;
00795                     insert_left<Node>(head->next[i], B);
00796                     rebalance<Node>(head, i);
00797                     return true;
00798                 }
00799                 else if(x + 1 == A.upper())
00800                 {
00801                     A.upper() = x;
00802                     return true;
00803                 }
00804             }
00805             bool rval = remove_scalar_H<Node>(head->next[i], x);
00806             rebalance<Node>(head, i);
00807             return rval;
00808         }
00809     }
00810 
00811     template <typename Node>
00812     bool remove_scalar(Pointer<Node>& head, Scalar<Node> data)
00813     {
00814         if(head == nullptr)
00815         {
00816             return false;
00817         }
00818 
00819         bool rval = remove_scalar_H<Node>(head, data);
00820 
00821         if(head->k == Node::N)
00822         {
00823             Pointer<Node> nn = new Node();
00824             nn->k = 0;
00825             nn->next[0] = head;
00826             rebalance<Node>(nn, 0);
00827             head = nn;
00828         }
00829         else if(head->k == 0)
00830         {
00831             Pointer<Node> tmp = head;
00832             head = head->next[0];
00833             delete tmp;
00834         }
00835 
00836         return rval;
00837     }
00838 
00839     template <typename Node>
00840     Scalar<Node> pop_scalar(Pointer<Node>& head)
00841     {
00842         if(head)
00843         {
00844             Scalar<Node> x = head->data[0].lower();
00845             remove_scalar<Node>(head, x);
00846             return x;
00847         }
00848         exit(-1);
```

```
00849     }
00850
00851     template <typename Node>
00852     void destruct(Pointer<Node> head)
00853     {
00854         if(head == nullptr)
00855         {
00856             return;
00857         }
00858         else
00859         {
00860             if(head->next[0] != nullptr)
00861             {
00862                 for(std::size_t i = 0; i < head->k; ++i)
00863                 {
00864                     destruct<Node>(head->next[i]);
00865                 }
00866             }
00867             delete head;
00868         }
00869     }
00870
00871
00872     template <typename Node>
00873     Data<Node> check_order(Pointer<Node> head, Data<Node> curr)
00874     {
00875         if(head != nullptr)
00876         {
00877             if(head->next[0] == nullptr)
00878             {
00879                 for(std::size_t i = 0; i < head->k; ++i)
00880                 {
00881                     if(curr > head->data[i])
00882                     {
00883                         std::cout « "ORDER WRONG!!!   --   " « curr « " > " « head->data[i] «
      std::endl;
00884                         exit(1);
00885                     }
00886                     curr = head->data[i];
00887                 }
00888             }
00889             else
00890             {
00891                 for(std::size_t i = 0; i < head->k; ++i)
00892                 {
00893                     curr = check_order<Node>(head->next[i], curr);
00894                     if(curr > head->data[i])
00895                     {
00896                         std::cout « "ORDER WRONG!!!   --   " « curr « " > " « head->data[i] «
      std::endl;
00897                         exit(1);
00898                     }
00899                     curr = head->data[i];
00900                 }
00901                 curr = check_order<Node>(head->next[head->k], curr);
00902             }
00903         }
00904         return curr;
00905     }
00906 } // End namespace index_tracker_detail
00907
00908 template <typename T, std::size_t d>
00909 std::ostream& operator«(std::ostream& out, const index_tracker_detail::BTreeNode<T,d>* head)
00910 {
00911     if(head == nullptr)
00912     {
00913         out « "[nil]";
00914     }
00915     else
00916     {
00917         out « "[ ";
00918         for(std::size_t i = 0; i < head->k; ++i)
00919         {
00920             if(head->next[0] != nullptr)
00921                 out « head->next[i] « " ";
00922             out « head->data[i] « " ";
00923         }
00924         if(head->next[0] != nullptr)
00925             out « head->next[head->k];
00926         out « "]";
00927     }
00928     return out;
00929 }
00930
00931 /**
00932  * @brief      Tracker of available indices implemented as a B-tree of intervals.
00933  *
```

```
00934  * @tparam     _T    Typename of the indices
00935  * @tparam     _d    Max number of interval bins = 2*value+1
00936  */
00937 template <typename _T, std::size_t _d = 16>
00938 class index_tracker
00939 {
00940 public:
00941     /// Typedef of BTree Node
00942     using Node = index_tracker_detail::BTreeNode<_T, _d>;
00943     using T = _T; /// Typename of the type to store
00944     constexpr static std::size_t d = _d; /// Number of bins
00945 
00946     /**
00947      * @brief       Initialize with interval [0~max)
00948      */
00949     index_tracker()
00950         : head(new Node(index_tracker_detail::Interval<T>(0,std::numeric_limits<T>::max())))
00951     {}
00952     ~index_tracker()
00953     {
00954         index_tracker_detail::destruct<Node>(head);
00955     }
00956 
00957     void insert(T x)
00958     {
00959         head = index_tracker_detail::insert_scalar<Node>(head, x);
00960     }
00961 
00962     index_tracker_detail::Scalar<Node> pop()
00963     {
00964         auto x = index_tracker_detail::pop_scalar<Node>(head);
00965         return x;
00966     }
00967 
00968     void remove(index_tracker_detail::Scalar<Node> x)
00969     {
00970         index_tracker_detail::remove_scalar<Node>(head, x);
00971     }
00972 
00973     bool empty() const
00974     {
00975         return head == nullptr;
00976     }
00977 
00978     friend std::ostream& operator<<(std::ostream& out, const index_tracker& x)
00979     {
00980         out << x.head;
00981         return out;
00982     }
00983 
00984 private:
00985     index_tracker_detail::Pointer<Node> head;
00986 };
00987 } // end namespace index_tracker
```

10.9 include/casc/Orientable.h File Reference

Data type for orientability.

```
#include <iostream>
#include <queue>
#include <set>
```

Data Structures

- struct casc::Orientable

  Class representing the orientation.

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<typename Complex >
  void casc::init_orientation (Complex &F)

  Initialize the partial ordering of the simplex edges.
- template<typename Complex >
  void casc::clear_orientation (Complex &F)

  Clear the orientation of the facets.
- template<typename Complex >
  std::tuple< int, bool, bool > casc::compute_orientation (Complex &F)

  Initializes and calculates the orientation of a simplicial_complex.
- template<typename Complex >
  std::tuple< int, bool, bool > casc::check_orientation (Complex &F)

  Checks for self consistent orientation and fill in missing orientations.

10.10 Orientable.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ******************************************************************************
00023  */
00024
00025 /**
00026  * @file  Orientable.h
00027  * @brief Data type for orientability
00028  */
00029
00030
00031 #pragma once
00032
00033 #include <iostream>
00034 #include <queue>
00035 #include <set>
00036
00037
00038 namespace casc{
00039 /**
00040  * @brief      Class representing the orientation.
00041  */
00042 struct Orientable {
00043     /// Integer representing +/- 1 orientation.
00044     int orientation;
00045 };
00046
00047 /// @cond detail
00048 /// Namespace for orientation helpers
00049 namespace orientation_detail{
00050
00051 template <class Complex, class SizeT >
00052 struct init_orientation_helper {};
00053
00054 template <class Complex, std::size_t k >
00055 struct init_orientation_helper<Complex, std::integral_constant<std::size_t, k»
```

```
00056 {
00057     static void f(Complex& F)
00058     {
00059         for(auto curr : F.template get_level_id<k>())
00060         {
00061             for(auto a : F.get_cover(curr))
00062             {
00063                 int orient = 1;
00064                 for(auto b : F.get_name(curr))
00065                 {
00066                     // Count the number of indices > name
00067                     if(a > b)
00068                     {
00069                         orient *= -1;
00070                     }
00071                     else
00072                     {
00073                         break;
00074                     }
00075                 }
00076                 (*F.get_edge_up(curr,a)).orientation = orient;
00077             }
00078         }
00079 
00080         init_orientation_helper<Complex,std::integral_constant<std::size_t,k+1»::f(F);
00081     }
00082 };
00083 
00084 /**
00085  * @brief      Terminating case for initializing orientation
00086  *
00087  * @tparam     Complex  Typename of the simplicial complex
00088  */
00089 template <typename Complex>
00090 struct init_orientation_helper<Complex, std::integral_constant<std::size_t, Complex::topLevel»
00091 {
00092     static void f(Complex&) {}
00093 };
00094 } // end namespace orientation_detail
00095 /// @endcond
00096 
00097 /**
00098  * @brief      Initialize the partial ordering of the simplex edges
00099  *
00100  * @param      F        Simplicial complex of interest
00101  *
00102  * @tparam     Complex  Typename of the simplicial complex
00103  */
00104 template <typename Complex>
00105 void init_orientation(Complex& F)
00106 {
00107     orientation_detail::init_orientation_helper<Complex,std::integral_constant<std::size_t,0»::f(F);
00108 }
00109 
00110 /**
00111  * @brief      Clear the orientation of the facets
00112  *
00113  * @param      F        Simplicial complex of interest
00114  *
00115  * @tparam     Complex  Typename of the simplicial complex
00116  */
00117 template <typename Complex>
00118 void clear_orientation(Complex& F)
00119 {
00120     // clear orientation
00121     for(auto& curr : F.template get_level<Complex::topLevel>())
00122     {
00123         curr.orientation = 0;
00124     }
00125 }
00126 
00127 // TODO: Implement this as a disjoint set operation during insertion (2)
00128 /**
00129  * @brief      Initializes and calculates the orientation of a
00130  *             simplicial_complex.
00131  *
00132  * @param      F        Simplicial_complex
00133  *
00134  * @tparam     Complex  Typename of the simplicial_complex.
00135  *
00136  * @return     A tuple of the number of connected components, where the complex
00137  *             is orientable, and if it is psuedo manifold.
00138  */
00139 template <typename Complex>
00140 std::tuple<int, bool, bool> compute_orientation(Complex& F)
00141 {
00142     init_orientation(F);
```

```
00143     clear_orientation(F);
00144     return check_orientation(F);
00145 }
00146 
00147 
00148 /**
00149  * @brief      Checks for self consistent orientation and fill in missing
00150  *             orientations
00151  *
00152  * @param      F        Simplicial_complex
00153  *
00154  * @tparam     Complex  Typename of the simplicial_complex.
00155  *
00156  * @return     A tuple of the number of connected components, where the complex
00157  *             is orientable, and if it is psuedo manifold.
00158  */
00159 template <typename Complex>
00160 std::tuple<int, bool, bool> check_orientation(Complex& F)
00161 {
00162     // compute orientation
00163     constexpr std::size_t k = Complex::topLevel - 1;
00164 
00165     std::deque<typename Complex::template SimplexID<k>> frontier;
00166     std::set<typename Complex::template SimplexID<k>> visited;
00167     int connected_components = 0;
00168     bool orientable = true;
00169     bool psuedo_manifold = true;
00170     for(auto outer : F.template get_level_id<k>())
00171     {
00172         if(visited.find(outer) == visited.end())
00173         {
00174             ++connected_components;
00175             frontier.push_back(outer);
00176 
00177             while(!frontier.empty())
00178             {
00179                 typename Complex::template SimplexID<k> curr = frontier.front();
00180                 if(visited.find(curr) == visited.end())
00181                 {
00182                     visited.insert(curr);
00183 
00184                     auto w = F.get_cover(curr);
00185 
00186                     if(w.size() == 1)
00187                     {
00188                         // w is a boundary
00189                         //std::cout << curr << ":" << w[0] << " ~ Boundary" << std::endl;
00190                     }
00191                     else if(w.size() == 2)
00192                     {
00193                         auto& edge0 = *F.get_edge_up(curr, w[0]);
00194                         auto& edge1 = *F.get_edge_up(curr, w[1]);
00195 
00196                         auto& node0 = *F.get_simplex_up(curr, w[0]);
00197                         auto& node1 = *F.get_simplex_up(curr, w[1]);
00198 
00199                         // If node0 doesn't have an orientation yet... Assign one
00200                         if(node0.orientation == 0)
00201                         {
00202                             if(node1.orientation == 0)
00203                             {
00204                                 node0.orientation = -1;
00205                                 node1.orientation = -edge1.orientation * edge0.orientation *
      node0.orientation;
00206                             }
00207                             else
00208                             {
00209                                 node0.orientation = -edge0.orientation * edge1.orientation *
      node1.orientation;
00210                             }
00211                         }
00212                         else
00213                         {
00214                             // if node1 doesn't have an orientation...
00215                             if(node1.orientation == 0)
00216                             {
00217                                 node1.orientation = -edge1.orientation * edge0.orientation *
      node0.orientation;
00218                             }
00219                             else
00220                             {
00221                                 // Check if the orientations are consistent
00222                                 if(edge0.orientation*node0.orientation +
      edge1.orientation*node1.orientation != 0)
00223                                 {
00224                                     orientable = false;
00225                                     // std::cout << "+++++" << std::endl;
```

```
00226                                   // std::cout « edge0.orientation « " : " « node0.orientation «
      std::endl;
00227                                   // std::cout « edge1.orientation « " : " « node1.orientation «
      std::endl;
00228
00229                                   // std::cout « " : "
00230                                   //           « edge0.orientation*node0.orientation +
      edge1.orientation*node1.orientation
00231                                   //           « std::endl;
00232                                   // std::cout « "-----"
00233                                   //           « std::endl;
00234                                   // std::cout « "Non-Orientable: "
00235                                   //           « edge0.orientation*node0.orientation +
      edge1.orientation*node1.orientation
00236                                   //           « std::endl;
00237                                 }
00238                               }
00239                           }
00240                           neighbors_up(F, curr, std::back_inserter(frontier));
00241                       }
00242                       else
00243                       {
00244                           // W.size() != 1 or 2
00245                           psuedo_manifold = false;
00246                       }
00247                   }
00248                   frontier.pop_front();
00249               }
00250           }
00251       }
00252       return std::make_tuple(connected_components, orientable, psuedo_manifold);
00253   }
00254   } // end namespace casc
```

10.11 include/casc/SimplexMap.h File Reference

SimplexMap data structure and associated convenience functions.

```
#include <array>
#include <map>
#include "util.h"
#include "stringutil.h"
```

Data Structures

- struct casc::SimplexMap< Complex >

  A multimap to represent a map of simplex indices to a set of simplices.

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<std::size_t k, typename Complex >
  static auto & casc::get (SimplexMap< Complex > &S)

  Get the map for a simplex dimension.

- template<std::size_t k, typename Complex >
  static auto & **casc::get** (const SimplexMap< Complex > &S)

  This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.

10.12 SimplexMap.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ******************************************************************************
00023  */
00024
00025 /**
00026  * @file  SimplexMap.h
00027  * @brief SimplexMap data structure and associated convenience functions.
00028  */
00029
00030 #pragma once
00031
00032 #include <array>
00033 #include <map>
00034 #include "util.h"
00035 #include "stringutil.h"
00036
00037 namespace casc
00038 {
00039
00040 /**
00041  * @brief      A multimap to represent a map of simplex indices to a set of
00042  *             simplices.
00043  *
00044  * @tparam     Complex  Typename of the simplicial_complex.
00045  */
00046 template <typename Complex>
00047 struct SimplexMap
00048 {
00049     /// Alias for SimplexID
00050     template <std::size_t j>
00051     using SimplexID = typename Complex::template SimplexID<j>;
00052     /// Index sequence of types from the simplicial_complex
00053     using LevelIndex = typename Complex::LevelIndex;
00054     /// Index sequence starting at 1
00055     using cLevelIndex = typename util::remove_first_val<std::size_t,
00056                                                         LevelIndex>::type;
00057     /// Reversed Index sequence
00058     using RevIndex = typename util::reverse_sequence<std::size_t,
00059                                                      LevelIndex>::type;
00060     /// Reversed index sequence stops at 1
00061     using cRevIndex = typename util::reverse_sequence<std::size_t,
00062                                                       cLevelIndex>::type;
00063     /// Typename of this object
00064     using type_this = SimplexMap<Complex>;
00065
00066     /**
00067      * @brief      Default constructor.
00068      */
00069     SimplexMap() {};
00070
00071     // TODO: Put in convenience functions for easy accession etc... (0)
00072     /**
00073      * @brief      Get the map for a particular simplex dimension.
00074      *
00075      * @tparam     k     Simplex dimension to retrieve.
00076      *
00077      * @return     A map of SimplexID<k> to SimplexSet.
00078      */
00079     template <std::size_t k>
00080     inline auto &get()
00081     {
00082         return std::get<k>(tupleMap);
```

```
00083     }
00084
00085     /**
00086      * @overload
00087      */
00088     template <std::size_t k>
00089     inline auto &get() const
00090     {
00091         return std::get<k>(tupleMap);
00092     }
00093
00094     /**
00095      * @brief      Print the SimplexMap.
00096      *
00097      * @param      output  Handle to the stream to print to.
00098      * @param[in]  S       SimplexMap to print.
00099      *
00100      * @return     Handle to the stream.
00101      */
00102     friend std ::ostream &operator«(std::ostream &output, const SimplexMap<Complex> &S)
00103     {
00104         output « "SimplexMap(";
00105         util::int_for_each<std::size_t, LevelIndex>(PrintHelper(),
00106                                                     output, S);
00107         output « ")";
00108         return output;
00109     }
00110
00111     private:
00112         /**
00113          * @brief      Helper struct to print the SimplexMap.
00114          */
00115         struct PrintHelper
00116         {
00117             /**
00118              * @brief      Print the SimplexMap.
00119              *
00120              * @param      output  Handle to the stream to print to.
00121              * @param[in]  S       SimplexMap to print.
00122              *
00123              * @tparam     k       The simplex dimension to print.
00124              */
00125             template <std::size_t k>
00126             static void apply(std::ostream &output, const SimplexMap<Complex> &S)
00127             {
00128                 output « "[l=" « k;
00129                 auto s = std::get<k>(S.tupleMap);
00130                 for (auto simplex : s)
00131                 {
00132                     output « ", " « to_string(simplex.first) « ":" « simplex.second;
00133                 }
00134                 output « "]";
00135             }
00136         };
00137
00138         /// Alias to create an Array of size k to store keys.
00139         template <std::size_t k> using array = std::array<typename Complex::KeyType, k>;
00140         /// A tuple of arrays of increasing size.
00141         using ArrayLevel = typename util::int_type_map<std::size_t, std::tuple, LevelIndex,
      array>::type;
00142         /// Alias for a Map of type T to a SimplexSet.
00143         template <class T> using map = std::map<T, SimplexSet<Complex> >;
00144         /// The full tuple of maps of an Array of keys to SimplexSet.
00145         typename util::type_map<ArrayLevel, map>::type tupleMap;
00146 };
00147
00148 /**
00149  * @brief      Get the map for a simplex dimension.
00150  *
00151  * @param      S        SimplexMap to retrieve from.
00152  *
00153  * @tparam     k        Simplex dimension.
00154  * @tparam     Complex  Typename of the complex.
00155  *
00156  * @return     Returns a map of std::Array<KeyType, k> to SimplexSet.
00157  */
00158 template <std::size_t k, typename Complex>
00159 static inline auto &get(SimplexMap<Complex> &S)
00160 {
00161     return S.template get<k>();
00162 }
00163
00164 /// @overload
00165 template <std::size_t k, typename Complex>
00166 static inline auto &get(const SimplexMap<Complex> &S)
00167 {
00168     return S.template get<k>();
```

```
00169 }
00170 } // end namespace casc
```

10.13 include/casc/SimplexSet.h File Reference

SimplexSet data structure and associated convenience functions.

```
#include <algorithm>
#include <unordered_set>
#include "util.h"
```

Data Structures

- struct casc::SimplexSet< Complex >

  A multiset to store simplices in a simplicial_complex.

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<std::size_t k, typename Complex >
  static auto & casc::get (SimplexSet< Complex > &S)

  Get the NodeSet for a simplex dimension from a SimplexSet.
- template<std::size_t k, typename Complex >
  static auto & casc::get (const SimplexSet< Complex > &S)
- template<typename Complex >
  bool casc::operator== (const SimplexSet< Complex > &lhs, const SimplexSet< Complex > &rhs)

  Compare if the sets are equivalent.
- template<typename Complex >
  bool casc::operator!= (const SimplexSet< Complex > &lhs, const SimplexSet< Complex > &rhs)

  Compare if the sets are not equivalent.
- template<typename Complex >
  static void casc::set_union (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

  Compute the set union.
- template<typename Complex >
  static void casc::set_intersection (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

  Compute the set intersection.
- template<typename Complex >
  static void casc::set_difference (const SimplexSet< Complex > &A, const SimplexSet< Complex > &B, SimplexSet< Complex > &dest)

  Compute the set difference.

10.14 SimplexSet.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ******************************************************************************
00023  */
00024
00025 /**
00026  * @file  SimplexSet.h
00027  * @brief SimplexSet data structure and associated convenience functions.
00028  */
00029
00030 #pragma once
00031
00032 #include <algorithm>
00033 #include <unordered_set>
00034 #include "util.h"
00035
00036 namespace casc
00037 {
00038
00039 /**
00040  * @brief      A multiset to store simplices in a simplicial_complex.
00041  *
00042  * This is really a tuple of sets where each set corresponds to a simplex
00043  * dimension. Many convenience functions are wrapped so this behaves much like
00044  * a std::set.
00045  *
00046  * @tparam     Complex  Typename of the simplicial_complex.
00047  */
00048 template <typename Complex>
00049 struct SimplexSet
00050 {
00051     /// Alias for SimplexID
00052     template <std::size_t j>
00053     using SimplexID = typename Complex::template SimplexID<j>;
00054     /// Index sequence of types from the simplicial_complex
00055     using LevelIndex = typename Complex::LevelIndex;
00056     /// Index sequence starting at 1
00057     using cLevelIndex = typename util::remove_first_val<std::size_t,
00058                                                         LevelIndex>::type;
00059     /// Reversed index sequence
00060     using RevIndex = typename util::reverse_sequence<std::size_t,
00061                                                      LevelIndex>::type;
00062     /// Reversed index sequence stops at 1
00063     using cRevIndex = typename util::reverse_sequence<std::size_t,
00064                                                       cLevelIndex>::type;
00065     /// Typename of this
00066     using type_this = SimplexSet<Complex>;
00067
00068     /// Tuple of SimplexIDs wrt an integral level.
00069     using SimplexIDLevel = typename util::int_type_map<std::size_t,
00070                                                        std::tuple, LevelIndex, SimplexID>::type;
00071     // No real sense to hide this tuple of sets from the end users.
00072     // Making it private, we'd have to introduce lots of friend structs.
00073     /// Tuple of NodeSets per level.
00074     typename util::type_map<SimplexIDLevel, NodeSet>::type tupleSet;
00075
00076     /// Default constructor
00077     SimplexSet() {};
00078     /// Default destructor
00079     ~SimplexSet() {};
00080
00081     // type_this& operator=(const type_this& other){
00082     //     util::int_for_each<std::size_t, LevelIndex>(CopyHelper(), this, other);
```

```
00083     // }
00084
00085     // type_this& operator=(type_this&& other){
00086     //     util::int_for_each<std::size_t, LevelIndex>(CopyHelper(), this, other);
00087     // }
00088
00089     /**
00090      * @brief      Checks if a level has no elements.
00091      *
00092      * @tparam     k     Level to check.
00093      *
00094      * @return     True if the container is empty, false otherwise.
00095      */
00096     template <std::size_t k>
00097     inline auto empty() const noexcept{
00098         return std::get<k>(tupleSet).empty();
00099     }
00100
00101     /**
00102      * @brief      Return the number of elements in a level.
00103      *
00104      * @tparam     k     Simplex dimension to query
00105      *
00106      * @return     Returns the number of simplices of dimension `k` are in the
00107      *             set.
00108      */
00109     template <std::size_t k>
00110     inline auto size() const noexcept{
00111         return std::get<k>(tupleSet).size();
00112     }
00113
00114     /**
00115      * @brief      Clear the contents.
00116      */
00117     void clear()
00118     {
00119         util::int_for_each<std::size_t, LevelIndex>(ClearHelper(), this);
00120     }
00121
00122     /**
00123      * @brief      Insert a simplex into the set.
00124      *
00125      * @param[in]  s     Simplex to insert.
00126      *
00127      * @tparam     k     Simplex dimension of 's'.
00128      */
00129     template <std::size_t k>
00130     inline void insert(SimplexID<k> s)
00131     {
00132         std::get<k>(tupleSet).insert(s);
00133     }
00134
00135     /**
00136      * @brief      Insert a SimplexSet into this.
00137      *
00138      * @param[in]  s     The SimplexSet to insert.
00139      */
00140     void insert(const SimplexSet<Complex> &s)
00141     {
00142         util::int_for_each<std::size_t, LevelIndex>(
00143             InsertHelper(), this, s);
00144     }
00145
00146     /**
00147      * @brief      Remove a simplex from the set.
00148      *
00149      * @param[in]  s     Simplex to remove.
00150      *
00151      * @tparam     k     Simplex dimension of 's'.
00152      */
00153     template <std::size_t k>
00154     inline void erase(SimplexID<k> s)
00155     {
00156         std::get<k>(tupleSet).erase(s);
00157     }
00158
00159     /**
00160      * @brief      Remove a set of simplices.
00161      *
00162      * @param[in]  s     SimplexSet to remove.
00163      */
00164     void erase(const SimplexSet<Complex> &s)
00165     {
00166         util::int_for_each<std::size_t, LevelIndex>(
00167             EraseHelper(), this, s);
00168     }
00169
```

```
00170     /**
00171      * @brief      Get the simplex of interest.
00172      *
00173      * @param[in]  s     The simplex to search for.
00174      *
00175      * @tparam     k     Simplex dimension of 's'.
00176      *
00177      * @return     Iterator to an element with key equivalent to s. If no such
00178      *             element is found, past-the-end iterator (see end()) is
00179      *             returned.
00180      */
00181     template <std::size_t k>
00182     inline auto find(const SimplexID<k> s)
00183     {
00184         return std::get<k>(tupleSet).find(s);
00185     }
00186
00187     /**
00188      * @brief      Get the simplex of interest.
00189      *
00190      * @param[in]  s     The simplex to search for.
00191      *
00192      * @tparam     k     Simplex dimension of 's'.
00193      *
00194      * @return     Iterator to an element with key equivalent to s. If no such
00195      *             element is found, past-the-end iterator (see end()) is
00196      *             returned.
00197      */
00198     template <std::size_t k>
00199     inline auto find(const SimplexID<k> s) const
00200     {
00201         return std::get<k>(tupleSet).find(s);
00202     }
00203
00204     /**
00205      * @brief      Get the past-the-end iterator.
00206      *
00207      * @tparam     k     The simplex dimension to get iterator of.
00208      *
00209      * @return     Returns an iterator to the element following the last element
00210      *             of the set for the specified simplex dimension.
00211      */
00212     template <std::size_t k>
00213     inline auto end()
00214     {
00215         return std::get<k>(tupleSet).end();
00216     }
00217
00218     /**
00219      * @brief      Get the past-the-end iterator.
00220      *
00221      * @tparam     k     The simplex dimension to get iterator of.
00222      *
00223      * @return     Returns an iterator to the element following the last element
00224      *             of the set for the specified simplex dimension.
00225      */
00226     template <std::size_t k>
00227     inline auto cend() const
00228     {
00229         return std::get<k>(tupleSet).cend();
00230     }
00231
00232     /**
00233      * @brief      Get an interator to the first element of the container.
00234      *
00235      * @tparam     k    The simplex dimension to get iterator of.
00236      *
00237      * @return     Returns an iterator to the first element.
00238      */
00239     template <std::size_t k>
00240     inline auto begin()
00241     {
00242         return std::get<k>(tupleSet).begin();
00243     }
00244
00245     /**
00246      * @brief      Get an interator to the first element of the container.
00247      *
00248      * @tparam     k    The simplex dimension to get iterator of.
00249      *
00250      * @return     Returns an iterator to the first element.
00251      */
00252     template <std::size_t k>
00253     inline auto cbegin() const
00254     {
00255         return std::get<k>(tupleSet).cbegin();
00256     }
```

```
00257
00258     // /**
00259     //  * @brief      Get the NodeSet for a particular simplex dimension.
00260     //  *
00261     //  * @tparam     k     Simplex dimension to get.
00262     //  *
00263     //  * @return     Returns the NodeSet corresponding to the requested dimension.
00264     //  */
00265     template <std::size_t k>
00266     inline auto &get()
00267     {
00268         return std::get<k>(tupleSet);
00269     }
00270
00271     // /**
00272     //  * @brief      Get the NodeSet for a particular simplex dimension.
00273     //  *
00274     //  * @tparam     k     Simplex dimension to get.
00275     //  *
00276     //  * @return     Returns the NodeSet corresponding to the requested dimension.
00277     //  */
00278     template <std::size_t k>
00279     inline auto &get() const
00280     {
00281         return std::get<k>(tupleSet);
00282     }
00283
00284     /**
00285      * @brief      Print the SimplexSet.
00286      *
00287      * See also casc::simplicial_complex::SimplexID::operator«.
00288      *
00289      * @param      output  Handle to the stream to print to.
00290      * @param[in]  S       SimplexSet to print.
00291      *
00292      * @return     Handle to the stream.
00293      */
00294     friend std ::ostream &operator«(std::ostream &output, const SimplexSet<Complex> &S)
00295     {
00296         output « "SimplexSet(";
00297         util::int_for_each<std::size_t, LevelIndex>(PrintHelper(),
00298                                                     output, S);
00299         output « ")";
00300         return output;
00301     }
00302
00303
00304     private:
00305         /**
00306          * @brief      Helper struct to insert a SimplexSet.
00307          */
00308         struct InsertHelper
00309         {
00310             /**
00311              * @brief      Perform the insertion for a dimension.
00312              *
00313              * @param      that  Typename of this SimplexSet.
00314              * @param[in]  S     SimplexSet to insert
00315              *
00316              * @tparam     k     Simplex dimension to insert.
00317              */
00318             template <std::size_t k>
00319             static void apply(type_this* that, const SimplexSet<Complex> &S)
00320             {
00321                 auto s = std::get<k>(S.tupleSet);
00322                 for (auto simplex : s)
00323                 {
00324                     that->insert(simplex);
00325                 }
00326             }
00327         };
00328
00329         /**
00330          * @brief      Helper struct to compute a set difference.
00331          */
00332         struct EraseHelper
00333         {
00334             /**
00335              * @brief      Perform the set difference for a dimension.
00336              *
00337              * @param      that  Typename of this SimplexSet.
00338              * @param[in]  S     SimplexSet to remove from this.
00339              *
00340              * @tparam     k     Simplex dimension to erase.
00341              */
00342             template <std::size_t k>
00343             static void apply(type_this* that, const SimplexSet<Complex> &S)
```

```
00344             {
00345                 auto s = std::get<k>(S.tupleSet);
00346                 for (auto simplex : s)
00347                 {
00348                     that->erase(simplex);
00349                 }
00350             }
00351         };
00352 
00353         /**
00354          * @brief      Helper struct to print the SimplexSet
00355          */
00356         struct PrintHelper
00357         {
00358             /**
00359              * @brief      Print the simplices in the level.
00360              *
00361              * @param      output  Handle to the stream to output to.
00362              * @param[in]  S       SimplexSet to print.
00363              *
00364              * @tparam     k       Simplex dimension to print.
00365              */
00366             template <std::size_t k>
00367             static void apply(std::ostream &output, const SimplexSet<Complex> &S)
00368             {
00369                 output « "[l=" « k;
00370                 auto s = std::get<k>(S.tupleSet);
00371                 for (auto simplex : s)
00372                 {
00373                     output « ", " « simplex;
00374                 }
00375                 output « "]";
00376             }
00377         };
00378 
00379         /**
00380          * @brief      Helper struct to clear the SimplexSet.
00381          */
00382         struct ClearHelper
00383         {
00384             /**
00385              * @brief      Clear a dimension.
00386              *
00387              * @param      that  Typename of this SimplexSet.
00388              *
00389              * @tparam     k     Simplex dimension to clear.
00390              */
00391             template <std::size_t k>
00392             void apply(type_this* that)
00393             {
00394                 auto &s = std::get<k>(that->tupleSet);
00395                 s.clear();
00396             }
00397         };
00398 
00399         // struct CopyHelper
00400         // {
00401         //     template <std::size_t k>
00402         //     void apply(type_this& that, type_this& other){
00403         //         auto &s = that.get<k>();
00404         //         s = other.get<k>();
00405         //     }
00406 
00407         //     template <std::size_t k>
00408         //     void apply(type_this& that, type_this&& other){
00409         //         auto &s = that.get<k>();
00410         //         s = other.get<k>();
00411         //     }
00412         // };
00413 };
00414 
00415 /**
00416  * @brief      Get the NodeSet for a simplex dimension from a SimplexSet.
00417  *
00418  * @param      S        SimplexSet of interest.
00419  *
00420  * @tparam     k        Simplex dimension desired.
00421  * @tparam     Complex  Typename of the simplicial_complex.
00422  *
00423  * @return     A NodeSet which holds simplices of dimension 'k' and a member of
00424  *             SimplexSet 'S'.
00425  */
00426 template <std::size_t k, typename Complex>
00427 static inline auto &get(SimplexSet<Complex> &S)
00428 {
00429     return S.template get<k>();
00430 }
```

```
00431
00432 /**
00433  * @overload
00434  */
00435 template <std::size_t k, typename Complex>
00436 static inline auto &get(const SimplexSet<Complex> &S)
00437 {
00438     return S.template get<k>();
00439 }
00440
00441 /// @cond detail
00442 /// Namespace for simplex container related helpers
00443 namespace simplex_set_detail
00444 {
00445
00446 /**
00447  * @brief      Helper struct to compute the union of two SimplexSets.
00448  *
00449  * @tparam     Complex  Typename of the simplicial_complex.
00450  */
00451 template <typename Complex>
00452 struct UnionH
00453 {
00454     /**
00455      * @brief      Compute the union of two SimplexSets.
00456      *
00457      * \f$A\cup B\f$
00458      *
00459      * @param[in]  A     A SimplexSet
00460      * @param[in]  B     Another SimplexSet
00461      * @param[out] dest  The destination SimplexSet
00462      *
00463      * @tparam     k     The current simplex dimension to merge.
00464      */
00465     template <std::size_t k>
00466     static void apply(const SimplexSet<Complex> &A,
00467                       const SimplexSet<Complex> &B,
00468                       SimplexSet<Complex>       &dest)
00469     {
00470         auto  a = std::get<k>(A.tupleSet);
00471         auto  b = std::get<k>(B.tupleSet);
00472         auto &d = std::get<k>(dest.tupleSet);
00473         d.insert(a.begin(), a.end());
00474         d.insert(b.begin(), b.end());
00475     }
00476 };
00477
00478 /**
00479  * @brief      Helper struct to compute the intersection of two SimplexSets.
00480  *
00481  * @tparam     Complex  Typename of the simplicial_complex.
00482  */
00483 template <typename Complex>
00484 struct IntersectH
00485 {
00486     /**
00487      * @brief      Compute the intersection of two SimplexSets.
00488      *
00489      * \f$A\cap B\f$
00490      *
00491      * @param[in]  A     A SimplexSet
00492      * @param[in]  B     Another SimplexSet
00493      * @param      dest  The destination SimplexSet.
00494      *
00495      * @tparam     k     The current simplex dimension to merge.
00496      */
00497     template <std::size_t k>
00498     static void apply(const SimplexSet<Complex> &A,
00499                       const SimplexSet<Complex> &B,
00500                       SimplexSet<Complex>       &dest)
00501     {
00502         auto  a = casc::get<k>(A);
00503         auto  b = casc::get<k>(B);
00504         auto &d = casc::get<k>(dest);
00505
00506         if (a.size() < b.size())
00507         {
00508             for (auto item : a)
00509             {
00510                 if (b.find(item) != b.end())
00511                     d.insert(item);
00512             }
00513         }
00514         else
00515         {
00516             for (auto item : b)
00517             {
```

```
00518                 if (a.find(item) != a.end())
00519                     d.insert(item);
00520             }
00521         }
00522     }
00523 };
00524
00525 /**
00526  * @brief      Helper struct to compute the set intersection.
00527  *
00528  * @tparam     Complex  Typename of the simplicial_complex.
00529  */
00530 template <typename Complex>
00531 struct DifferenceH
00532 {
00533     /**
00534      * @brief      Compute the set difference for a simplex dimension.
00535      *
00536      * \f$ dest = A \setminus B \f$
00537      *
00538      * @param[in]  A     A SimplexSet.
00539      * @param[in]  B     Remove this SimplexSet from A.
00540      * @param      dest  The destination SimplexSet.
00541      *
00542      * @tparam     k     The simplex dimension to compute the difference of.
00543      */
00544     template<std::size_t k>
00545     static void apply(const SimplexSet<Complex> &A,
00546                       const SimplexSet<Complex> &B,
00547                       SimplexSet<Complex>       &dest)
00548     {
00549         auto  a = casc::get<k>(A);
00550         auto  b = casc::get<k>(B);
00551         auto &d = casc::get<k>(dest);
00552
00553         for (auto item : a)
00554         {
00555             if (b.find(item) == b.end())
00556                 d.insert(item);
00557         }
00558     }
00559 };
00560
00561 /**
00562  * @brief      Helper struct to compute set equivalence.
00563  *
00564  * @tparam     Complex  Typename of the simplicial_complex.
00565  */
00566 template <typename Complex>
00567 struct OperatorEQH
00568 {
00569     /// Result of the comparison
00570     bool result;
00571
00572     /// Default constructor
00573     OperatorEQH(): result(true) {}
00574
00575     /**
00576      * @brief      Compare the two sets by level.
00577      *
00578      * @param[in]  lhs   The left hand side
00579      * @param[in]  rhs   The right hand side
00580      *
00581      * @tparam     k     Level to compare.
00582      */
00583     template <std::size_t k>
00584     void apply(const SimplexSet<Complex> &lhs,
00585                       const SimplexSet<Complex> &rhs){
00586         auto a = casc::get<k>(lhs);
00587         auto b = casc::get<k>(rhs);
00588         result &= a==b;
00589     }
00590 };
00591 }     // end namespace simplex_set_detail
00592 /// @endcond
00593
00594 /**
00595  * @brief      Compare if the sets are equivalent
00596  *
00597  * @param[in]  lhs      The left hand side
00598  * @param[in]  rhs      The right hand side
00599  *
00600  * @tparam     Complex  Typename of the simplicial_complex
00601  *
00602  * @return     True if the sets are equal, false otherwise.
00603  */
00604 template <typename Complex>
```

```
00605 bool operator==(const SimplexSet<Complex> &lhs, const SimplexSet<Complex> &rhs){
00606     auto func = simplex_set_detail::OperatorEQH<Complex>();
00607     util::int_for_each<std::size_t, typename Complex::LevelIndex>(
00608         func, lhs, rhs);
00609     return func.result;
00610 }
00611
00612 /**
00613  * @brief      Compare if the sets are not equivalent.
00614  *
00615  * @param[in]  lhs      The left hand side
00616  * @param[in]  rhs      The right hand side
00617  *
00618  * @tparam     Complex  Typename of the simplicial_complex.
00619  *
00620  * @return     True if the sets are inequal, false otherwise.
00621  */
00622 template <typename Complex>
00623 bool operator!=(const SimplexSet<Complex> &lhs, const SimplexSet<Complex> &rhs){
00624     return !(lhs == rhs);
00625 }
00626
00627 /**
00628  * @brief      Compute the set union.
00629  *
00630  * @param[in]  A        A SimplexSet
00631  * @param[in]  B        Another SimplexSet
00632  * @param[out] dest     The destination SimplexSet.
00633  *
00634  * @tparam     Complex  Typename of the simplicial_complex.
00635  */
00636 template <typename Complex>
00637 static void set_union(const SimplexSet<Complex> &A,
00638                       const SimplexSet<Complex> &B,
00639                       SimplexSet<Complex>       &dest)
00640 {
00641     util::int_for_each<std::size_t,
00642                        typename Complex::LevelIndex>(
00643         simplex_set_detail::UnionH<Complex>(), A, B, dest);
00644 }
00645
00646 /**
00647  * @brief      Compute the set intersection.
00648  *
00649  * @param[in]  A        A SimplexSet
00650  * @param[in]  B        Another SimplexSet
00651  * @param[out] dest     The destination SimplexSet.
00652  *
00653  * @tparam     Complex  Typename of the simplicial_complex.
00654  */
00655 template <typename Complex>
00656 static void set_intersection(const SimplexSet<Complex> &A,
00657                              const SimplexSet<Complex> &B,
00658                              SimplexSet<Complex>       &dest)
00659 {
00660     util::int_for_each<std::size_t,
00661                        typename Complex::LevelIndex>(
00662         simplex_set_detail::IntersectH<Complex>(), A, B, dest);
00663 }
00664
00665 /**
00666  * @brief      Compute the set difference.
00667  *
00668  * @param[in]  A        A SimplexSet
00669  * @param[in]  B        Another SimplexSet
00670  * @param[out] dest     The destination SimplexSet.
00671  *
00672  * @tparam     Complex  Typename of the simplicial_complex.
00673  */
00674 template <typename Complex>
00675 static void set_difference(const SimplexSet<Complex> &A,
00676                            const SimplexSet<Complex> &B,
00677                            SimplexSet<Complex>       &dest)
00678 {
00679     util::int_for_each<std::size_t,
00680                        typename Complex::LevelIndex>(
00681         simplex_set_detail::DifferenceH<Complex>(), A, B, dest);
00682 }
00683 } // end namespace casc
```

10.15 include/casc/SimplicialComplex.h File Reference

This header contains the main CASC data structure and associated components.

```
#include <algorithm>
#include <assert.h>
#include <cstdint>
#include <map>
#include <set>
#include <iterator>
#include <array>
#include <vector>
#include <iostream>
#include <fstream>
#include <functional>
#include <type_traits>
#include <ostream>
#include <unordered_set>
#include <unordered_map>
#include <utility>
#include <stdexcept>
#include "index_tracker.h"
#include "util.h"
```

Data Structures

- class casc::simplicial_complex< traits >

  The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring.
- struct casc::simplicial_complex< traits >::SimplexID< k >

  A handle for a simplex object in the complex.
- struct casc::simplicial_complex< traits >::EdgeID< k >

  External reference to an edge or a connection within the complex.

Namespaces

- namespace casc

  Namespace for everything CASC.

Typedefs

- template<typename KeyType , typename ... Ts>
  using casc::AbstractSimplicialComplex = simplicial_complex< detail::simplicial_complex_traits_default< KeyType, Ts... > >
- template<typename T >
  using **casc::NodeSet** = std::unordered_set< T, simplex_set_detail::hashSimplexID< T > >

  Helpful alias defining a unordered_set of simplices. See also hashSimplexID.

10.16 SimplicialComplex.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ******************************************************************************
00023  */
00024
00025 /**
00026  * @file  SimplicialComplex.h
00027  * @brief This header contains the main CASC data structure and associated
00028  *        components.
00029  */
00030
00031 #pragma once
00032
00033 #include <algorithm>
00034 #include <assert.h>
00035 #include <cstdint>
00036 #include <map>
00037 #include <set>
00038 #include <iterator>
00039 #include <array>
00040 #include <vector>
00041 #include <iostream>
00042 #include <fstream>
00043 #include <functional>
00044 #include <type_traits>
00045 #include <ostream>
00046 #include <unordered_set>
00047 #include <unordered_map>
00048 #include <utility>
00049 #include <stdexcept>
00050
00051 #include "index_tracker.h"
00052 #include "util.h"
00053
00054 #if __has_cpp_attribute(maybe_unused)
00055 #define MAYBE_UNUSED [[maybe_unused]]
00056 #else
00057 #define MAYBE_UNUSED
00058 #endif
00059
00060 /// Namespace for everything CASC
00061 namespace casc
00062 {
00063 /// @cond detail
00064 /// Namespace for CASC internal data structures
00065 namespace detail
00066 {
00067 /// Data structure to store simplices by level.
00068 template <class T> using map = std::map<std::size_t, T>;
00069
00070 /**
00071  * @brief      A generic pair type representing Key to Value associations.
00072  *
00073  * @tparam     T1    Typename of the Key
00074  * @tparam     T2    Typename of the Value
00075  */
00076 template <typename T1, typename T2>
00077 struct asc_pair {
00078     using this_t = asc_pair<T1, T2>;
00079     asc_pair() {}
00080     asc_pair(const T1& first, const T2& second) : _pair(first, second) {}
00081     asc_pair(T1&& first, T2&& second) : _pair(std::forward<T1>(first), std::forward<T2>(second)) {}
00082     asc_pair(const this_t& other) : _pair(other._pair) {}
```

```
00083     asc_pair(this_t&& other) : _pair(std::forward<std::pair<T1,T2>>(other._pair)) {}
00084
00085     operator T1() const {
00086         return _pair.first;
00087     }
00088
00089     this_t& operator=(const this_t& other){
00090         _pair = other._pair;
00091         return *this;
00092     }
00093
00094     this_t& operator=(this_t&& other){
00095         _pair = std::move(other._pair);
00096         return *this;
00097     }
00098
00099     friend bool operator==(const this_t& lhs, const this_t& rhs) {return lhs.first == rhs.first;}
00100     friend bool operator!=(const this_t& lhs, const this_t& rhs) {return lhs.first != rhs.first;}
00101     friend bool operator<=(const this_t& lhs, const this_t& rhs) {return lhs.first <= rhs.first;}
00102     friend bool operator>=(const this_t& lhs, const this_t& rhs) {return lhs.first >= rhs.first;}
00103     friend bool operator<(const this_t& lhs, const this_t& rhs) {return lhs.first < rhs.first;}
00104     friend bool operator>(const this_t& lhs, const this_t& rhs) {return lhs.first > rhs.first;}
00105
00106     T1& first = _pair.first;
00107     T2& second = _pair.second;
00108 private:
00109     std::pair<T1,T2> _pair;
00110 };
00111
00112 /**
00113  * @brief      Array of asc_pairs sorted by Key for boundary adjacency storage.
00114  *
00115  * @tparam     KEY_T  Typename of Key
00116  * @tparam     VAL_T  Typename of Value
00117  * @tparam     k      Size of the array
00118  */
00119 template <typename KEY_T, typename VAL_T, std::size_t k>
00120 struct asc_arraymap {
00121     using pair_t = asc_pair<KEY_T, VAL_T>;
00122     using array_t = std::array<pair_t, k>;
00123     using iterator = typename array_t::iterator;
00124     using const_iterator = typename array_t::const_iterator;
00125
00126     asc_arraymap(){
00127         _begin = _array.begin();
00128         _end = _array.begin();
00129     }
00130
00131     void insert(pair_t& p){
00132         if (_end == _array.end())
00133             throw std::out_of_range("insert&: Adding element beyond the end of array.");
00134         *_end = p;
00135         ++_end;
00136         std::sort(_begin, _end);
00137     }
00138
00139     void insert(pair_t&& p){
00140         if (_end == _array.end())
00141             throw std::out_of_range("insert&&: Adding element beyond the end of array.");
00142         *_end = std::forward<pair_t>(p);
00143         ++_end;
00144         std::sort(_begin, _end);
00145     }
00146
00147     iterator find(const KEY_T& key){
00148         return std::find(_begin, _end, key);
00149     }
00150
00151     void erase(const KEY_T& key){
00152         auto it = std::find(_begin, _end, key);
00153         if(it != _end){
00154             std::copy(it+1, _end, it);
00155             --_end;
00156         }
00157     }
00158
00159     std::size_t size() const{
00160         return std::distance(_end, _begin);
00161     }
00162
00163     VAL_T& operator[](const KEY_T& key){
00164         auto it = std::find(_begin, _end, key);
00165         if(it != _end){
00166             return it->second;
00167         }
00168         else{
00169             if (_end == _array.end())
```

```
00170                 throw std::out_of_range("operator[]: Adding element beyond the end of array.");
00171             _end->first = key;
00172             ++_end;
00173             std::sort(_begin, _end);
00174             return std::find(_begin,_end, key)->second;
00175         }
00176     }
00177
00178     iterator begin(){ return _begin; }
00179     iterator end(){ return _end; }
00180     const_iterator cbegin() const {return _begin;}
00181     const_iterator cend() const {return _end;}
00182
00183 private:
00184     array_t _array;
00185     iterator _begin;
00186     iterator _end;
00187 };
00188
00189
00190 /**
00191  * @brief      Sorted vector of asc_pairs for coboundary relation storage.
00192  *
00193  * @tparam     KEY_T  Typename of Key
00194  * @tparam     VAL_T  Typename of Values
00195  */
00196 template <typename KEY_T, typename VAL_T>
00197 struct asc_vectormap {
00198     using pair_t = asc_pair<KEY_T, VAL_T>;
00199     using vector_t = std::vector<pair_t>;
00200     using iterator = typename vector_t::iterator;
00201     using const_iterator = typename vector_t::const_iterator;
00202
00203     asc_vectormap(){}
00204
00205     void insert(pair_t& p){
00206         iterator first = std::lower_bound(_vector.begin(), _vector.end(), p);
00207         if ((first == _vector.end()) || (*first != p)){
00208             _vector.insert(first, p);
00209         }
00210         else{
00211             std::cout « "Item already exists...";
00212         }
00213     }
00214
00215     void insert(pair_t&& p){
00216         iterator first = std::lower_bound(_vector.begin(), _vector.end(), p);
00217         if ((first == _vector.end()) || (*first != p)){
00218             _vector.insert(first, std::forward<pair_t>(p));
00219         }
00220         else{
00221             std::cout « "Item already exists...";
00222         }
00223     }
00224
00225     iterator find(const KEY_T& key){
00226         iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
00227         if (first != _vector.end()){
00228             if (*first != key){
00229                 return _vector.end();
00230             }
00231             else{
00232                 return first;
00233             }
00234         }
00235         else{
00236             return first;
00237         }
00238     }
00239
00240     void erase(const KEY_T& key){
00241         iterator it = this->find(key);
00242         if (it != _vector.end()){
00243             _vector.erase(it);
00244         }
00245     }
00246
00247     std::size_t size() const{
00248         return _vector.size();
00249     }
00250
00251     VAL_T& at(const KEY_T& key){
00252         iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
00253         if ((first == _vector.end()) || (first->first != key)){
00254             throw std::out_of_range("Could not find element in asc_vectormap.");
00255         }
00256         else {
```

```
00257             return first->second;
00258         }
00259     }
00260
00261     VAL_T& operator[](const KEY_T& key){
00262         iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
00263         if ((first == _vector.end()) || (first->first != key)){
00264             first = _vector.emplace(first, pair_t());
00265             first->first = key;
00266             return first->second;
00267         }
00268         else {
00269             return first->second;
00270         }
00271     }
00272
00273     iterator begin(){ return _vector.begin(); }
00274     iterator end(){ return _vector.end(); }
00275     const_iterator cbegin() const {return _vector.cbegin();}
00276     const_iterator cend() const {return _vector.cend();}
00277 private:
00278     vector_t _vector;
00279 };
00280
00281
00282 /**
00283  * @brief Template prototype for Nodes in CASC.
00284  *
00285  * asc_Node must be defined outside of simplicial_complex because C++ does
00286  * not allow internal templates to be partially specialized. This template
00287  * prototype is later specialized to represent various Node roles.
00288  */
00289 template <class KeyType, std::size_t k, std::size_t N, typename DataTypes, class> struct asc_Node;
00290
00291 /// This is the base Node class.
00292 struct asc_NodeBase {
00293     /**
00294      * @brief      Construct a Node
00295      *
00296      * @param[in]  id    An internal integer identifier of the Node.
00297      */
00298     asc_NodeBase(std::size_t id) : _node(id) {}
00299     virtual ~asc_NodeBase() {}; /**< Destructor */
00300     std::size_t _node;              /**< Internal Node ID*/
00301 };
00302
00303 /**
00304  * @brief      Base class for Node with some data.
00305  *
00306  * @tparam     DataType  Typename of the data to be stored.
00307  */
00308 template <class DataType>
00309 struct asc_NodeData {
00310     DataType _data;     /**< stored data with type DataType */
00311 };
00312
00313 /**
00314  * @brief      Explicit specialization for Nodes without data.
00315  *
00316  * This exists so that the compiler knows to not allocate any memory to
00317  * store data when void is specified.
00318  */
00319 template <>
00320 struct asc_NodeData<void> {};
00321
00322 /**
00323  * @brief      Base class for Nodes with edge data.
00324  *
00325  * @tparam     KeyType   Typename of index for indexing Nodes.
00326  * @tparam     DataType  Typename of the data stored on the edge.
00327  */
00328 template <class KeyType, class DataType>
00329 struct asc_EdgeData {
00330     /** The map of SimplexIDs to stored edge data. */
00331     std::unordered_map<KeyType, DataType> _edge_data;
00332 };
00333
00334 /**
00335  * @brief      Explicit specialization for Nodes with no edge data.
00336  *
00337  * @tparam     KeyType   Typename of index for indexing Nodes.
00338  */
00339 template <class KeyType>
00340 struct asc_EdgeData<KeyType, void> {};
00341
00342 /**
00343  * @brief      Base class for Node with parent nodes
```

```
00344  *
00345  * @tparam     KeyType        Typename of the Node index
00346  * @tparam     k              The Simplex dimension
00347  * @tparam     N              Dimension of the Complex
00348  * @tparam     NodeDataTypes  A util::type_holder array of Node types
00349  * @tparam     EdgeDataTypes  A util::type_holder array of Edge types
00350  */
00351 template <  class KeyType,
00352             std::size_t k,
00353             std::size_t N,
00354             class NodeDataTypes,
00355             class EdgeDataTypes>
00356 struct asc_NodeDown :
00357     public asc_EdgeData<KeyType,
00358                         typename util::type_get<k-1, EdgeDataTypes>::type> {
00359     /** Alias the typename of the parent Node */
00360     using DownNodeT = asc_Node<KeyType, k-1, N, NodeDataTypes, EdgeDataTypes>;
00361
00362     /** Map of indices to parent Node pointers*/
00363     asc_arraymap<KeyType, DownNodeT*, k> _down;
00364     // std::map<KeyType, DownNodeT*> _down;
00365 };
00366
00367 /**
00368  * @brief      Base class for Node with children Nodes
00369  *
00370  * @tparam     KeyType        Typename of the Node index
00371  * @tparam     k              The Simplex dimension
00372  * @tparam     N              Dimension of the Complex
00373  * @tparam     NodeDataTypes  A util::type_holder array of Node types
00374  * @tparam     EdgeDataTypes  A util::type_holder array of Edge types
00375  */
00376 template <  class KeyType,
00377             std::size_t k,
00378             std::size_t N,
00379             class NodeDataTypes,
00380             class EdgeDataTypes>
00381 struct asc_NodeUp {
00382     /// Typename of the nodes up.
00383     using UpNodeT = asc_Node<KeyType, k+1, N, NodeDataTypes, EdgeDataTypes>;
00384     asc_vectormap<KeyType, UpNodeT*> _up;
00385     // std::unordered_map<KeyType, UpNodeT*> _up;      /**< @brief Map of pointers to children */
00386 };
00387
00388 /**
00389  * @brief      Node with both parents and children
00390  *
00391  * @tparam     KeyType        Typename of the Node index
00392  * @tparam     k              The Simplex dimension
00393  * @tparam     N              Dimension of the Complex
00394  * @tparam     NodeDataTypes  A util::type_holder of Node types
00395  * @tparam     EdgeDataTypes  A util::type_holder of Edge types
00396  */
00397 template <class KeyType, std::size_t k, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
00398 struct asc_Node : public asc_NodeBase,
00399                   public asc_NodeData<typename util::type_get<k, NodeDataTypes>::type>,
00400                   public asc_NodeDown<KeyType, k, N, NodeDataTypes, EdgeDataTypes>,
00401                   public asc_NodeUp<KeyType, k, N, NodeDataTypes, EdgeDataTypes>
00402 {
00403     /// Dimension of the simplex.
00404     static constexpr std::size_t level = k;
00405
00406     /**
00407      * @brief      Default constructor
00408      *
00409      * @param[in]  id    The internal integer identifier.
00410      */
00411     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00412
00413     /**
00414      * @brief      Print the Node out for debugging only
00415      *
00416      * @param      output  The output stream.
00417      * @param[in]  node    The Node of interest to print.
00418      *
00419      * @return     A handle to the output stream.
00420      */
00421     friend std ::ostream &operator«(std::ostream &output, const asc_Node &node)
00422     {
00423         output « "Node(level=" « k « ", " « "id=" « node._node;
00424         if (node._down.size() > 0)
00425         {
00426             for (auto it = node._down.cbegin(); it != node._down.cend(); ++it)
00427             {
00428                 output « ", NodeDownID={'"
00429                        « it->first « "', "
00430                        « it->second->_node « "}";
```

```
00431             }
00432         }
00433         if (node._up.size() > 0)
00434         {
00435             for (auto it = node._up.cbegin(); it != node._up.cend(); ++it)
00436             {
00437                 output « ", NodeUpID={'"
00438                        « it->first « "', "
00439                        « it->second->_node « "}";
00440             }
00441         }
00442         output « ")";
00443         return output;
00444     }
00445 };
00446
00447 /**
00448  * @brief      Node with only children i.e., the root.
00449  *
00450  * @tparam     KeyType        Typename of the Node index
00451  * @tparam     N              The Simplex dimension
00452  * @tparam     NodeDataTypes  A util::type_holder of Node types
00453  * @tparam     EdgeDataTypes  A util::type_holder of Edge types
00454  */
00455 template <class KeyType, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
00456 struct asc_Node<KeyType, 0, N, NodeDataTypes, EdgeDataTypes> :
00457     public asc_NodeBase,
00458     public asc_NodeData<typename util::type_get<0, NodeDataTypes>::type>,
00459     public asc_NodeUp<KeyType, 0, N, NodeDataTypes, EdgeDataTypes>
00460 {
00461     /// Dimension of the simplex.
00462     static constexpr std::size_t level = 0;
00463
00464     /**
00465      * @brief      Default constructor
00466      *
00467      * @param[in]  id    The internal integer identifier.
00468      */
00469     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00470
00471     /**
00472      * @brief      Print the Node out for debugging only
00473      *
00474      * @param      output  The output stream.
00475      * @param[in]  node    The Node of interest to print.
00476      *
00477      * @return     A handle to the output stream.
00478      */
00479     friend std ::ostream &operator«(std::ostream &output, const asc_Node &node)
00480     {
00481         output « "Node(level=" « 0
00482                « ", id=" « node._node;
00483         if (node._up.size() > 0)
00484         {
00485             for (auto it = node._up.cbegin(); it != node._up.cend(); ++it)
00486             {
00487                 output « ", NodeUpID={'"
00488                        « it->first « "', "
00489                        « it->second->_node « "}";
00490             }
00491         }
00492         output « ")";
00493         return output;
00494     }
00495 };
00496
00497 /**
00498  * @brief      Top level node with only parents
00499  *
00500  * @tparam     KeyType        Typename of the Node index
00501  * @tparam     N              The Simplex dimension
00502  * @tparam     NodeDataTypes  A util::type_holder of Node types
00503  * @tparam     EdgeDataTypes  A util::type_holder of Edge types
00504  */
00505 template <class KeyType, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
00506 struct asc_Node<KeyType, N, N, NodeDataTypes, EdgeDataTypes> :
00507     public asc_NodeBase,
00508     public asc_NodeData<typename util::type_get<N, NodeDataTypes>::type>,
00509     public asc_NodeDown<KeyType, N, N, NodeDataTypes, EdgeDataTypes>
00510 {
00511     /// Dimension of the simplex.
00512     static constexpr std::size_t level = N;
00513
00514     /**
00515      * @brief      Default constructor
00516      *
00517      * @param[in]  id    The internal integer identifier.
```

```
00518      */
00519     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00520
00521     /**
00522      * @brief      Print the Node out for debugging only
00523      *
00524      * @param      output  The output stream.
00525      * @param[in]  node    The Node of interest to print.
00526      *
00527      * @return     A handle to the output stream.
00528      */
00529     friend std ::ostream &operator«(std::ostream &output, const asc_Node &node)
00530     {
00531         output « "Node(level=" « N
00532                « ", id=" « node._node;
00533         if (node._down.size() > 0)
00534         {
00535             for (auto it = node._down.cbegin(); it != node._down.cend(); ++it)
00536             {
00537                 output « ", NodeDownID={'"
00538                        « it->first « "', "
00539                        « it->second->_node « "}";
00540             }
00541         }
00542         output « ")";
00543         return output;
00544     }
00545 };
00546
00547 /**
00548  * @brief      An iterator adapter to iterate over NodeIDs.
00549  *
00550  * @tparam     Iter  Typename of the iterator
00551  * @tparam     Data  Typename of the data
00552  */
00553 template <typename Iter, typename Data>
00554 struct node_id_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
00555     public:
00556         /// Inherit from a bidirectional std::iterator.
00557         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
00558         /// Empty constructor
00559         node_id_iterator() {}
00560         /// Instantiate with an iterator to wrap
00561         node_id_iterator(Iter j) : i(j) {}
00562         /// Increment the iterator
00563         node_id_iterator &operator++() { ++i; return *this; }
00564         /// Increment the iterator
00565         node_id_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
00566         /// Decrement the iterator
00567         node_id_iterator &operator--() { --i; return *this; }
00568         /// Decrement hte iterator
00569         node_id_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
00570         /// Iterator equality comparison
00571         bool operator==(node_id_iterator j) const { return i == j.i; }
00572         /// Iterator inequality comparison
00573         bool operator!=(node_id_iterator j) const { return !(*this == j); }
00574         /// Dereferencing the iterator produces a SimplexID.
00575         Data operator*() { return Data(i->second); }
00576         /// Const version
00577         const Data operator*() const {return Data(i->second); }
00578         /// Dereferencing the iterator produces a SimplexID.
00579         typename super::pointer operator->() { return Data(i->second); }
00580     protected:
00581         /// The iterator to wrap.
00582         Iter i;
00583 };
00584
00585 /**
00586  * @brief      Convert an iterator into a node_id_iterator.
00587  *
00588  * @param[in]  j     The iterator to wrap
00589  *
00590  * @tparam     Iter  Typename of the iterator
00591  * @tparam     Data  Typename of the data
00592  *
00593  * @return     An iterator over NodeIDs.
00594  */
00595 template <typename Iter, typename Data>
00596 inline node_id_iterator<Iter, Data> make_node_id_iterator(Iter j)
00597 {
00598     return node_id_iterator<Iter, Data>(j);
00599 }
00600
00601 /**
00602  * @brief      An iterator adapter to iterate over Node data.
00603  *
00604  * @tparam     Iter  Typename of the iterator
```

```
00605  * @tparam     Data  Typename of the data
00606  */
00607 template <typename Iter, typename Data>
00608 struct node_data_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
00609     public:
00610         /// Inherit from a bidirectional std::iterator.
00611         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
00612         /// Empty constructor.
00613         node_data_iterator() {}
00614         /// Instantiate with an iterator to wrap.
00615         node_data_iterator(Iter j) : i(j) {}
00616         /// Increment the iterator
00617         node_data_iterator &operator++() { ++i; return *this; }
00618         /// Increment the iterator
00619         node_data_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
00620         /// Decrement the iterator
00621         node_data_iterator &operator--() { --i; return *this; }
00622         /// Decrement the iterator
00623         node_data_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
00624         /// Iterator comparison
00625         bool operator==(node_data_iterator j) const { return i == j.i; }
00626         /// Iterator inequality comparison
00627         bool operator!=(node_data_iterator j) const { return !(*this == j); }
00628         /// Dereferencing the iterator produces the data.
00629         typename super::reference operator*() { return i->second->_data; }
00630         /// Dereferencing the iterator produces the data.
00631         typename super::pointer operator->() { return i->second->_data; }
00632     protected:
00633         /// The wrapped iterator.
00634         Iter i;
00635 };
00636
00637 /**
00638  * @brief      Convert an iterator into a node_data_iterator.
00639  *
00640  * @param[in]  j     The iterator to wrap
00641  *
00642  * @tparam     Iter  Typename of the iterator
00643  * @tparam     Data  Typename of the data
00644  *
00645  * @return     An iterator over Node data.
00646  */
00647 template <typename Iter, typename Data>
00648 inline node_data_iterator<Iter, Data> make_node_data_iterator(Iter j)
00649 {
00650     return node_data_iterator<Iter, Data>(j);
00651 }
00652
00653 /**
00654  * @brief      Helper to build a traits struct via expanding explicitly
00655  * specified
00656  *             traits from AbstractSimplicialComplex.
00657  *
00658  * @tparam     K     Typename for the KeyType
00659  * @tparam     Ts    Types of data to be stored on simplices.
00660  */
00661 template <typename K, typename ... Ts>
00662 struct simplicial_complex_traits_default
00663 {
00664     /// Template to assign ints for all levels.
00665     template <std::size_t k> using all_int = int;
00666     /// Alias for KeyType
00667     using KeyType   = K;
00668     /// The typenames of the data to be stored on simplices.
00669     using NodeTypes = util::type_holder<Ts...>;
00670     /// Assign all_int type to all edges
00671     using EdgeTypes = typename util::int_type_map<std::size_t,
00672                                                   util::type_holder,
00673                                                   typename std::make_index_sequence<sizeof ...
      (Ts)-1>,
00674                                                   all_int>::type;
00675 };
00676 } // end namespace detail
00677 /// @endcond
00678
00679 /**
00680  * @class      simplicial_complex
00681  *
00682  * @brief      The CASC data structure for representing simplicial complexes of
00683  *             arbitrary dimensionality with coloring.
00684  *
00685  * You can create a CASC object by defining a struct containing the
00686  * traits of the complex. For example:
00687  * ~~~~~~~~~~~~~~~{.cpp}
00688  * struct complex_traits{
00689  *     using KeyType = int;
00690  *     using NodeTypes = util::type_holder<int,int,int,int>;
```

```
00691  *     using EdgeTypes = util::type_holder<int,int,int>;
00692  * };
00693  *
00694  * using SurfaceMesh = simplicial_complex<complex_traits>;
00695  * ~~~~~~~~~~~~~~~~
00696  * This is the preferred method for creating a new CASC type. Alternatively you
00697  * can use the ::AbstractSimplicialComplex alias to build a struct for you.
00698  *
00699  * @tparam     traits  A struct defining the dimension of the complex and data
00700  *                     to be stored on each node and edge.
00701  */
00702 template <typename traits>
00703 class simplicial_complex
00704 {
00705     public:
00706         /// Typename of simplex keys.
00707         using KeyType = typename traits::KeyType;
00708         /// Typenames of the data stored on simplices.
00709         using NodeDataTypes = typename traits::NodeTypes;
00710         /// Typenames of the data stored on edges.
00711         using EdgeDataTypes = typename traits::EdgeTypes;
00712         /// Type of this
00713         using type_this = simplicial_complex<traits>;
00714         /// Total number of levels in the complex.
00715         static constexpr std::size_t numLevels = NodeDataTypes::size;
00716         /// Dimension of the simplicial complex.
00717         static constexpr std::size_t topLevel  = numLevels-1;
00718         /// Dimension of boundaries.
00719         static constexpr std::size_t bdryLevel  = numLevels-2;
00720         /// Index of all simplex dimensions in the complex.
00721         using LevelIndex = typename std::make_index_sequence<numLevels>;
00722     private:
00723         /// Alias templated asc_node<...> as Node<k>
00724         template <std::size_t k> using Node = detail::asc_Node<KeyType, k, topLevel, NodeDataTypes,
      EdgeDataTypes>;
00725         /// Alias Node<k>* as NodePtr<k>
00726         template <std::size_t k> using NodePtr = Node<k>*;
00727
00728     public:
00729         /** Convenience alias for the user specified NodeData<k> typename */
00730         template <std::size_t k> using NodeData = typename util::type_get<k, NodeDataTypes>::type;
00731         /** Convenience alias for the user specified EdgeData<k> typename */
00732         template <std::size_t k> using EdgeData = typename util::type_get<k, EdgeDataTypes>::type;
00733
00734         friend struct SimplexID; /**< SimplexID is a friend of
00735                                     simplicial_complex */
00736
00737         /**
00738          * @brief      A handle for a simplex object in the complex.
00739          *
00740          * SimplexID wraps a Node* for external handling. This way
00741          * the end users are never exposed to a raw pointer. For all general
00742          * purposes algorithms should use and pass SimplexIDs over raw pointers.
00743          *
00744          * @tparam     k     The Simplex dimension.
00745          */
00746         template <std::size_t k>
00747         struct SimplexID {
00748             /// Typename of the complex
00749             using complex = simplicial_complex<traits>;
00750             /// SimplexID is a friend of the complex
00751             friend simplicial_complex<traits>;
00752             /// The dimension of the simplex.
00753             static constexpr std::size_t level = k;
00754
00755             /**
00756              * @brief      Default constructor wraps a nullptr.
00757              */
00758             SimplexID() : ptr(nullptr) {}
00759
00760             /**
00761              * @brief      Constructor to wrap a NodePtr<k>.
00762              *
00763              * @param[in]  p     The NodePtr to wrap
00764              */
00765             SimplexID(NodePtr<k> p) : ptr(p) {}
00766
00767             /**
00768              * @brief      Copy constructor.
00769              *
00770              * @param[in]  rhs   Another SimplexID to copy.
00771              */
00772             SimplexID(const SimplexID &rhs) : ptr(rhs.ptr) {}
00773
00774             /// Assignment operator
00775             SimplexID &operator=(const SimplexID &rhs) { ptr = rhs.ptr; return *this;}
00776
```

```
00777             /// Equality of wrapped pointers
00778             friend bool operator==(SimplexID lhs, SimplexID rhs) { return lhs.ptr == rhs.ptr; }
00779             /// Inequality of wrapped pointers
00780             friend bool operator!=(SimplexID lhs, SimplexID rhs) { return lhs.ptr != rhs.ptr; }
00781             /// Compare wrapped pointers
00782             friend bool operator<=(SimplexID lhs, SimplexID rhs) { return lhs.ptr <= rhs.ptr; }
00783             /// Compare wrapped pointers
00784             friend bool operator>=(SimplexID lhs, SimplexID rhs) { return lhs.ptr >= rhs.ptr; }
00785             /// Compare wrapped pointers
00786             friend bool operator<(SimplexID lhs, SimplexID rhs)  { return lhs.ptr < rhs.ptr; }
00787             /// Compare wrapped pointers
00788             friend bool operator>(SimplexID lhs, SimplexID rhs)  { return lhs.ptr > rhs.ptr; }
00789
00790             /// Support casting to uintptr_t for hashing.
00791             explicit operator std::uintptr_t () const { return reinterpret_cast<std::uintptr_t>(ptr);
      }
00792
00793             /// Dereferencing a SimplexID returns the data stored.
00794             complex::NodeData<k> const &operator*() const { return ptr->_data; }
00795             /// Dereferencing a SimplexID returns the data stored.
00796             complex::NodeData<k> &operator*() { return ptr->_data; }
00797
00798             /// Get a handle to the stored data.
00799             complex::NodeData<k> const &data() const { return ptr->_data; }
00800             /// Get a handle to the stored data.
00801             complex::NodeData<k>        &data() { return ptr->_data; }
00802
00803             /**
00804              * @brief      Gets the name of a simplex as an std::Array.
00805              *
00806              * @param[in]  id    SimplexID of the simplex of interest.
00807              *
00808              * @return     Array containing the name of 'id'.
00809              */
00810             std::array<KeyType, k> indices() const
00811             {
00812                 std::array<KeyType, k> s;
00813                 std::size_t            i = 0;
00814                 for (auto curr : ptr->_down)
00815                 {
00816                     s[i++] = curr.first;
00817                 }
00818
00819                 return std::move(s);
00820             }
00821
00822             // Valid in C++17
00823             // TODO: (0) expose this to modern compilers
00824             // if constexpr (k < complex::topLevel){
00825                 /**
00826                  * @brief      Insert the coboundary keys of a simple into an inserter.
00827                  *
00828                  * @param[in]  pos       Iterator inserter
00829                  *
00830                  * @tparam     Inserter  Typename of the inserter.
00831                  */
00832                 template <class Inserter>
00833                 void cover_insert(Inserter pos) const
00834                 {
00835                     for (auto curr : ptr->_up)
00836                     {
00837                         *pos++ = curr.first;
00838                     }
00839                 }
00840
00841                 /**
00842                  * @brief      Get the coboundary keys of a simplex.
00843                  *
00844                  * @return     A vector of coboundary indices.
00845                  */
00846                 std::vector<KeyType> cover() const
00847                 {
00848                     std::vector<KeyType> rval;
00849                     cover_insert(std::back_inserter(rval));
00850                     return rval;
00851                 }
00852             // }
00853
00854             /**
00855              * @brief      Get a coboundary simplex
00856              *
00857              * @param[in]  s     Array of keys to follow
00858              *
00859              * @tparam     j     Number of keys
00860              *
00861              * @return     The simplex up
00862              */
```

```
00863             template <std::size_t j>
00864             SimplexID<k+j> get_simplex_up(const KeyType (&s)[j]) const
00865             {
00866                 static_assert(k+j <= complex::topLevel, "Cannot get simplex greater than the facets");
00867                 return complex::get_recurse<k, j>::apply(s, this->ptr);
00868             }
00869
00870             /**
00871              * @brief      Get a coboundary simplex
00872              *
00873              * @param[in]  arr   Array of keys to follow
00874              *
00875              * @tparam     j     Number of keys
00876              *
00877              * @return     The simplex up
00878              */
00879             template <std::size_t j>
00880             SimplexID<k+j> get_simplex_up(const std::array<KeyType, j> &arr) const
00881             {
00882                 static_assert(k+j <= complex::topLevel, "Cannot get simplex greater than the facets");
00883                 return get_recurse<k, j>::apply(arr.data(), this->ptr);
00884             }
00885
00886             /**
00887              * @brief      Convenience version of get_simplex_up when the name 's'
00888              *             consists of a single character.
00889              *
00890              * @param[in]  id    The identifier of a simplex.
00891              * @param[in]  s     The relative single character name of the desired
00892              *                   simplex.
00893              *
00894              * @tparam     i     The size of simplex 'id'.
00895              *
00896              * @return     SimplexID of node corresponding to \f$id\cup s\f$.
00897              */
00898             SimplexID<k+1> get_simplex_up(const KeyType s) const
00899             {
00900                 return get_recurse<k, 1>::apply(&s, this->ptr);
00901             }
00902
00903             /**
00904              * @brief      Gets the simplex down.
00905              */
00906             template <std::size_t j>
00907             SimplexID<k-j> get_simplex_down(const KeyType (&s)[j]) const
00908             {
00909                 return get_down_recurse<k, j>::apply(s, this->ptr);
00910             }
00911
00912             /**
00913              * @brief      Gets the simplex down.
00914              */
00915             template <std::size_t j>
00916             SimplexID<k-j> get_simplex_down(const std::array<KeyType, j> &arr) const
00917             {
00918                 return get_down_recurse<k, j>::apply(arr.data(), this->ptr);
00919             }
00920
00921             /**
00922              * @brief      Gets the simplex down.
00923              */
00924             SimplexID<k-1> get_simplex_down(const KeyType s) const
00925             {
00926                 return get_down_recurse<k, 1>::apply(&s, this->ptr);
00927             }
00928
00929             /**
00930              * @brief      Print the simplex as its name.
00931              *
00932              * @param      out   Handle to the stream
00933              * @param[in]  nid   SimplexID of interest
00934              *
00935              * @return     Handle to the stream
00936              *
00937              * Example
00938              * ~~~~~~~~~~~~~~~(.c)
00939              * mesh.insert<3>({0,1,2});
00940              * std::cout « s « std::endl;
00941              * s{0,1,2}"
00942              * ~~~~~~~~~~~~~~~
00943              */
00944             friend std ::ostream &operator«(std::ostream    &out,
00945                                             const SimplexID &nid)
00946             {
00947                 // currently no such thing as static_if in c++ so we use a
00948                 // template
00949                 // helper
```

```
00950                 out « "s{";
00951                 print_helper<k, 0>::apply(out, nid);
00952                 out « "}";
00953                 return out;
00954             }
00955
00956
00957             // NOTE: Manually swap out these print functions for debugging if
00958             // desired.
00959             // /**
00960             //  * @brief      A full debug printout of of the node itself
00961             //  *
00962             //  * @param      out   Handle to the stream
00963             //  * @param[in]  nid   SimplexID of interest
00964             //  *
00965             //  * @return     Handle to the stream
00966             //  */
00967             // friend std::ostream& operator«(std::ostream& out, const
00968             // SimplexID& nid){ out « *nid.ptr; return out; }
00969
00970             // /**
00971             //  * @brief      Print the SimplexID as an ID.
00972             //  *
00973             //  * Example "0x7fd502402f10"
00974             //  *
00975             //  * @param      out   Handle to the stream
00976             //  * @param[in]  nid   Node of interest
00977             //  *
00978             //  * @return     Handle to the stream
00979             //  */
00980             // friend std ::ostream &operator«(std::ostream &out, const
00981             // SimplexID &nid) { out « nid.ptr; return out; }
00982
00983             private:
00984                 /**
00985                  * @brief      Base Case helper for printing SimplexIDs.
00986                  *
00987                  * @tparam     l     The Simplex dimension
00988                  * @tparam     foo   Dummy argument to avoid explicit
00989                  * specialization
00990                  *                   in class scope
00991                  */
00992                 template <std::size_t l, std::size_t foo>
00993                 struct print_helper
00994                 {
00995                     /**
00996                      * @brief      Print out the name of the simplex.
00997                      *
00998                      * @param      out   Stream to pipe to.
00999                      * @param[in]  nid   The simplex to print.
01000                      *
01001                      * @return     Handle to the output stream.
01002                      */
01003                     static std::ostream &apply(std::ostream       &out,
01004                                                const SimplexID<l> &nid)
01005                     {
01006                         auto down = (*nid.ptr)._down;
01007                         for (auto it = down.cbegin(); it != down.cend()-1; ++it)
01008                         {
01009                             out « it->first « ",";
01010                         }
01011                         out « (down.cend()-1)->first;
01012                         return out;
01013                     }
01014                 };
01015
01016                 /**
01017                  * @brief      Explicit specialization to print 0-Simplices
01018                  *
01019                  * @tparam     foo   Dummy argument to avoid explicit
01020                  * specialization
01021                  *                   in class scope
01022                  */
01023                 template <std::size_t foo>
01024                 struct print_helper<0, foo>
01025                 {
01026                     /**
01027                      * @brief      Print the root simplex
01028                      *
01029                      * @param      out   Stream to print to.
01030                      * @param[in]  nid   The simplex to print.
01031                      *
01032                      * @return     Handle to the output stream.
01033                      */
01034                     static std::ostream &apply(std::ostream    &out,
01035                                                const SimplexID &nid)
01036                     {
```

```
01037                       out « "root " « nid;
01038                       return out;
01039                   }
01040               };
01041               /// The wrapped pointer.
01042               NodePtr<k> ptr;
01043       };
01044
01045       friend struct EdgeID; /**< EdgeID is a friend to simplicial_complex */
01046
01047       /**
01048        * @brief      External reference to an edge or a connection within the
01049        *             complex.
01050        *
01051        * @tparam     k     The edge connects a simplex of size k-1 to a
01052        *                   simplex of size k.
01053        */
01054       template <std::size_t k>
01055       struct EdgeID {
01056           /// Typename of the complex
01057           using complex = simplicial_complex<traits>;
01058           /// EdgeID is a friend of the complex
01059           friend simplicial_complex<traits>;
01060           /// The dimension of the simplex which the edge points to.
01061           static constexpr std::size_t level = k;
01062
01063           /**
01064            * @brief      Default contstructor wraps a nullptr and dummy edge.
01065            */
01066           EdgeID() : ptr(nullptr), edge(0) {}
01067
01068           /**
01069            * @brief      Constructor to wrap an Edge.
01070            *
01071            * @param[in]  p     Pointer to the next Node.
01072            * @param[in]  e     Key of the edge
01073            */
01074           EdgeID(NodePtr<k> p, KeyType e) : ptr(p), edge(e) {}
01075
01076           /**
01077            * @brief      Copy constructor
01078            *
01079            * @param[in]  rhs   The right hand side
01080            */
01081           EdgeID(const EdgeID &rhs) : ptr(rhs.ptr), edge(rhs.edge) {}
01082
01083           /// Assignment operator
01084           EdgeID &operator=(const EdgeID &rhs) { ptr = rhs.ptr; edge = rhs.edge; return *this;}
01085
01086           /// Equality of wrapped pointers and edges
01087           friend bool operator==(EdgeID lhs, EdgeID rhs) { return lhs.ptr == rhs.ptr && lhs.edge ==
      rhs.edge; }
01088           /// Compare wrapped pointers and edges.
01089           friend bool operator!=(EdgeID lhs, EdgeID rhs) { return !(lhs == rhs); }
01090           /// Compare wrapped pointers and edges.
01091           friend bool operator<=(EdgeID lhs, EdgeID rhs) { return lhs < rhs || lhs == rhs; }
01092           /// Compare wrapped pointers and edges.
01093           friend bool operator>=(EdgeID lhs, EdgeID rhs) { return lhs > rhs || lhs == rhs; }
01094           /// Less than defines an ordering of key types on the edges.
01095           friend bool operator<(EdgeID lhs, EdgeID rhs)
01096           {
01097               return (lhs.ptr < rhs.ptr) || (lhs.ptr == rhs.ptr && lhs.edge < rhs.edge);
01098           }
01099           /// Greater than comparison
01100           friend bool operator>(EdgeID lhs, EdgeID rhs)  { return rhs < lhs; }
01101
01102           // explicit operator std::size_t () const { return
01103           // static_cast<std::size_t>(ptr);
01104
01105           /// Dereferencing an EdgeID gets the data on the edge.
01106           auto const &operator*() const { return data(); }
01107           /// Dereferencing an EdgeID gets the data on the edge.
01108           auto &operator*() { return data(); }
01109
01110           /// Get the key of the edge.
01111           KeyType     key() const { return edge; }
01112
01113           /// Return the data stored on the edge.
01114           auto const &data() const { return ptr->_edge_data[edge]; }
01115           /// Return the data stored on the edge.
01116           auto       &data() { return ptr->_edge_data[edge]; }
01117
01118           /**
01119            * @brief      Get the coboundary simplex.
01120            *
01121            * @return     SimplexID of the simplex above the edge.
01122            */
```

```
01123             SimplexID<k> up() const { return ptr; }
01124
01125             /**
01126              * @brief      Get the simplex below.
01127              *
01128              * @return     SimplexID of the simplex below the edge.
01129              */
01130             SimplexID<k-1> down() const { return SimplexID<k-1>(ptr->_down[edge]); }
01131
01132             private:
01133                 /// Pointer to the next node.
01134                 NodePtr<k> ptr;
01135                 /// The Key of the edge.
01136                 KeyType    edge;
01137         };
01138
01139         /**
01140          * @brief      Default constructor
01141          */
01142         simplicial_complex()
01143             : node_count(0)
01144         {
01145             for (auto &x : level_count) // Initialize level_count to 0 for all
01146                                         // levels
01147             {
01148                 x = 0;
01149             }
01150             // Create a root node
01151             _root = create_node<0>();
01152         }
01153
01154         /**
01155          * @brief      Destruct the simplicial complex.
01156          *
01157          * Recursively go over the simplices and remove them prior to
01158          * destructing
01159          * the CASC object itself.
01160          */
01161         ~simplicial_complex()
01162         {
01163             std::size_t count;
01164             remove_recurse<0, 0>::apply(this, &_root, &_root + 1, count);
01165         }
01166
01167         /**
01168          * @brief      Insert a simplex and all sub-simplices into the complex.
01169          *
01170          * Example -- insert the simplex {1,2,3}:
01171          * ~~~~~~~~~~~~~~~{.cpp}
01172          * mesh.insert<3>({1,2,3});
01173          * ~~~~~~~~~~~~~~~
01174          *
01175          * @param[in]  s     A C style array of vertices of simplex 's'.
01176          *
01177          * @tparam     n     Dimension of simplex 's'.
01178          */
01179         template <std::size_t n>
01180         SimplexID<n> insert(const KeyType (&s)[n])
01181         {
01182             for (const KeyType* p = s; p < s + n; ++p)
01183             {
01184                 unused_vertices.remove(*p);
01185             }
01186             return insert_full<0, n>::apply(this, _root, s);
01187         }
01188
01189         /**
01190          * @brief      Insert a simplex and all sub-simplices into the complex
01191          *             along with data.
01192          *
01193          * Example -- insert the simplex {1,2,3} with data:
01194          * ~~~~~~~~~~~~~~~{.cpp}
01195          * mesh.insert<3>({1,2,3}, 5);
01196          * ~~~~~~~~~~~~~~~
01197          *
01198          * @param[in]  s     A C style array of vertices of simplex 's'.
01199          * @param[in]  data  The data to be stored at the simplex 's'.
01200          *
01201          * @tparam     n     Dimension of simplex 's'.
01202          */
01203         template <std::size_t n>
01204         SimplexID<n> insert(const KeyType (&s)[n], const NodeData<n> &data)
01205         {
01206             for (const KeyType* p = s; p < s + n; ++p)
01207             {
01208                 unused_vertices.remove(*p);
01209             }
```

```
01210             Node<n>* rval = insert_full<0, n>::apply(this, _root, s);
01211             rval->_data = data;
01212             return rval;
01213         }
01214 
01215         /**
01216          * @brief      Insert a simplex named and all sub-simplices into the
01217          * complex.
01218          *
01219          * @param[in]  s     Array of vertices comprising 's'.
01220          *
01221          * @tparam     n     Dimension of simplex 's'.
01222          */
01223         template <std::size_t n>
01224         SimplexID<n> insert(const std::array<KeyType, n> &s)
01225         {
01226             for (KeyType x : s)
01227             {
01228                 unused_vertices.remove(x);
01229             }
01230             return insert_full<0, n>::apply(this, _root, s.data());
01231         }
01232 
01233         /**
01234          * @brief      Insert a simplex and all sub-simplices into the complex
01235          *             along with data.
01236          *
01237          * @param[in]  s     Array of vertices comprising 's'.
01238          * @param[in]  data  The data to be stored at the simplex 's'.
01239          *
01240          * @tparam     n     Dimension of simplex 's'.
01241          */
01242         template <std::size_t n>
01243         SimplexID<n> insert(const std::array<KeyType, n> &s, const NodeData<n> &data)
01244         {
01245             for (KeyType x : s)
01246             {
01247                 unused_vertices.remove(x);
01248             }
01249             Node<n>* rval = insert_full<0, n>::apply(this, _root, s.data());
01250             rval->_data = data;
01251             return rval;
01252         }
01253 
01254         /**
01255          * @brief      Add a new vertex to the complex.
01256          *
01257          * A list of currently unused indices are tracked using a B-tree. This
01258          * function retrieves a currently unused index and creates a new vertex
01259          * while returning the new key.
01260          *
01261          * @return     The key of the new vertex.
01262          */
01263         KeyType add_vertex()
01264         {
01265             KeyType v[1] = {unused_vertices.pop()};
01266             insert<1>(v);
01267             return v[0];
01268         }
01269 
01270         /**
01271          * @brief      Add a new vertex to the complex with data.
01272          *
01273          * @return     The key of the new vertex.
01274          */
01275         KeyType add_vertex(const NodeData<1> &data)
01276         {
01277             KeyType v[1] = {unused_vertices.pop()};
01278             insert<1>(v, data);
01279             return v[0];
01280         }
01281 
01282 
01283         /**
01284          * @brief      Apply a lambda function the name of a simplex.
01285          *
01286          * @param[in]  id      SimplexID of the simplex of interest.
01287          * @param[in]  fn      Lambda function to apply to the name of 'id'.
01288          *
01289          * @tparam     n       Dimension of simplex 'id'.
01290          * @tparam     Lambda  Functor which supports operator(KeyType).
01291          */
01292         template <std::size_t n, typename Lambda>
01293         void get_name(SimplexID<n> id, Lambda fn) const
01294         {
01295             for (auto curr : id.ptr->_down)
01296             {
```

```
01297                 fn(curr.first);
01298             }
01299         }
01300 
01301         /**
01302          * @brief      Gets the name of a simplex as an std::Array.
01303          *
01304          * @param[in]  id    SimplexID of the simplex of interest.
01305          *
01306          * @tparam     n     Size of the simplex referenced by 'id'.
01307          *
01308          * @return     Array containing the name of 'id'.
01309          */
01310         template <std::size_t n>
01311         std::array<KeyType, n> get_name(SimplexID<n> id) const
01312         {
01313             std::array<KeyType, n> s;
01314             std::size_t            i = 0;
01315             for (auto curr : id.ptr->_down)
01316             {
01317                 s[i++] = curr.first;
01318             }
01319             assert(i == n);
01320             return s;
01321         }
01322 
01323         /**
01324          * @brief      Gets the name of a simplex.
01325          *
01326          * This is the explicit specialization which handles the empty set
01327          * simplex.
01328          *
01329          * @param[in]  id    SimplexID of the simplex of interest.
01330          *
01331          * @return     Array containing the name of 'id'.
01332          */
01333         std::array<KeyType, 0> get_name(SimplexID<0>) const
01334         {
01335             std::array<KeyType, 0> name{};
01336             return name;
01337         }
01338 
01339 
01340         /**
01341          * @brief      Gets the simplex with name 's'.
01342          *
01343          * @param[in]  s     Name of the simplex to find.
01344          *
01345          * @tparam     n     Dimension of simplex s.
01346          *
01347          * @return     SimplexID of node corresponding to 's'.
01348          */
01349         template <std::size_t n>
01350         SimplexID<n> get_simplex_up(const KeyType (&s)[n]) const
01351         {
01352             return get_recurse<0, n>::apply(s, _root);
01353         }
01354 
01355         template <std::size_t n>
01356         SimplexID<n> get_simplex_up(const std::array<KeyType, n> &arr) const
01357         {
01358             return get_recurse<0, n>::apply(arr.data(), _root);
01359         }
01360 
01361         /**
01362          * @brief      Get the simplex identifier which has the name 's'
01363          *             relative to the simplex 'id'.
01364          *
01365          * @param[in]  id    The identifier of a simplex.
01366          * @param[in]  s     The relative name of the desired simplex.
01367          *
01368          * @tparam     i     The size of simplex 'id'.
01369          * @tparam     j     The length of the name 's'.
01370          *
01371          * @return     SimplexID of node corresponding to \f$id\cup s\f$.
01372          */
01373         template <std::size_t i, std::size_t j>
01374         SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const KeyType (&s)[j]) const
01375         {
01376             return get_recurse<i, j>::apply(s, id);
01377         }
01378 
01379         template <std::size_t i, std::size_t j>
01380         SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const std::array<KeyType, j> &arr) const
01381         {
01382             return get_recurse<i, j>::apply(arr.data(), id);
01383         }
```

```
01384
01385
01386         /**
01387          * @brief      Convenience version of get_simplex_up when the name 's'
01388          *             consists of a single character.
01389          *
01390          * @param[in]  id    The identifier of a simplex.
01391          * @param[in]  s     The relative single character name of the desired
01392          *                   simplex.
01393          *
01394          * @tparam     i     The size of simplex 'id'.
01395          *
01396          * @return     SimplexID of node corresponding to \f$id\cup s\f$.
01397          */
01398         template <std::size_t i>
01399         SimplexID<i+1> get_simplex_up(const SimplexID<i> id, const KeyType s) const
01400         {
01401             return get_recurse<i, 1>::apply(&s, id.ptr);
01402         }
01403
01404         /**
01405          * @brief      Get the root simplex.
01406          *
01407          * @return     The root simplex.
01408          */
01409         SimplexID<0> get_simplex_up() const
01410         {
01411             return _root;
01412         }
01413
01414
01415         /**
01416          * @brief      Get the sub-simplex of the simplex 'id' which does not
01417          *             have 's' in the name.
01418          *
01419          * @param[in]  id    The identifier of a simplex.
01420          * @param[in]  s     The relative name of the desired simplex.
01421          *
01422          * @tparam     i     The size of simplex 'id'.
01423          * @tparam     j     The length of the name 's'
01424          *
01425          * @return     The node down.
01426          */
01427         template <std::size_t i, std::size_t j>
01428         SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const KeyType (&s)[j]) const
01429         {
01430             return get_down_recurse<i, j>::apply(s, id.ptr);
01431         }
01432
01433         template <std::size_t i, std::size_t j>
01434         SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const std::array<KeyType, j> &arr)
      const
01435         {
01436             return get_down_recurse<i, j>::apply(arr.data(), id.ptr);
01437         }
01438
01439         /**
01440          * @brief      Convenience version of get_simplex_down when the name 's'
01441          *             consists of a single character.
01442          *
01443          * @param[in]  id    The identifier of a simplex.
01444          * @param[in]  s     The relative single character name of the desired
01445          *                   simplex.
01446          *
01447          * @tparam     i     The size of simplex 'id'.
01448          *
01449          * @return     The node down.
01450          */
01451         template <std::size_t i>
01452         SimplexID<i-1> get_simplex_down(const SimplexID<i> id, const KeyType s) const
01453         {
01454             return get_down_recurse<i, 1>::apply(&s, id.ptr);
01455         }
01456
01457         /**
01458          * @brief      Get the root simplex.
01459          *
01460          * @return     The root simplex.
01461          */
01462         SimplexID<0> get_simplex_down() const
01463         {
01464             return _root;
01465         }
01466
01467         /**
01468          * @brief      Insert the coboundary keys of a simple into an inserter.
01469          *
```

```
01470           * @param[in]  id        The identifier of a simplex.
01471           * @param[in]  pos       Iterator inserter
01472           *
01473           * @tparam     k         The dimension of the simplex.
01474           * @tparam     Inserter  Typename of the inserter.
01475           */
01476          template <std::size_t k, class Inserter>
01477          void get_cover_insert(const SimplexID<k> id, Inserter pos) const
01478          {
01479              for (auto curr : id.ptr->_up)
01480              {
01481                  *pos++ = curr.first;
01482              }
01483          }
01484
01485          /**
01486           * @brief      Apply a lambda function to the coboundary keys.
01487           *
01488           * @param[in]  id      The identifier
01489           * @param[in]  fn      The function
01490           *
01491           * @tparam     k       The dimension of the simplex.
01492           * @tparam     Lambda  Typename of a functor which supports
01493           * operator(KeyType).
01494           */
01495          template <std::size_t k, class Lambda>
01496          void get_cover(const SimplexID<k> id, Lambda fn) const
01497          {
01498              for (auto curr : id.ptr->_up)
01499              {
01500                  fn(curr.first);
01501              }
01502          }
01503
01504          /**
01505           * @brief      Get the coboundary keys of a simplex.
01506           *
01507           * @param[in]  id    The identifier of a simplex.
01508           *
01509           * @tparam     k     The dimension of the simplex.
01510           *
01511           * @return     A vector of coboundary indices.
01512           */
01513          template <std::size_t k>
01514          std::vector<KeyType> get_cover(const SimplexID<k> id) const
01515          {
01516              std::vector<KeyType> rval;
01517              get_cover_insert(id, std::back_inserter(rval));
01518              return rval;
01519          }
01520
01521          /**
01522           * @brief      Get the coboundary of a set of simplices.
01523           *
01524           * @param      simplices  The set of simplices
01525           *
01526           * @tparam     k          The dimension of the simplices.
01527           *
01528           * @return     The set of coboundary simplices.
01529           */
01530          template <std::size_t k>
01531          std::set<SimplexID<k+1> > up(const std::set<SimplexID<k> > &&simplices) const
01532          {
01533              std::set<SimplexID<k+1> > rval;
01534              for (auto simplex : simplices)
01535              {
01536                  for (auto p : simplex.ptr->_up)
01537                  {
01538                      rval.insert(SimplexID<k+1>(p.second));
01539                  }
01540              }
01541              return rval;
01542          }
01543
01544          /**
01545           * @brief      Get the coboundary of a set of simplices.
01546           *
01547           * @param      simplices  The set of simplices
01548           *
01549           * @tparam     k          The dimension of the simplices.
01550           *
01551           * @return     The set of coboundary simplices.
01552           */
01553          template <std::size_t k>
01554          std::set<SimplexID<k+1> > up(const std::set<SimplexID<k> > &simplices) const
01555          {
01556              std::set<SimplexID<k+1> > rval;
```

```
01557             for (auto simplex : simplices)
01558             {
01559                 for (auto p : simplex.ptr->_up)
01560                 {
01561                     rval.insert(SimplexID<k+1>(p.second));
01562                 }
01563             }
01564             return rval;
01565         }
01566 
01567         /**
01568          * @brief      Get the coboundary of a simplex.
01569          *
01570          * @param      nid   The simplex of interest
01571          *
01572          * @tparam     k     The dimension of the simplex.
01573          *
01574          * @return     Set of (k+1)-simplices of which 'nid' is a face of.
01575          */
01576         template <std::size_t k>
01577         std::set<SimplexID<k+1> > up(const SimplexID<k> nid) const
01578         {
01579             std::set<SimplexID<k+1> > rval;
01580             for (auto p : nid.ptr->_up)
01581             {
01582                 rval.insert(SimplexID<k+1>(p.second));
01583             }
01584             return rval;
01585         }
01586 
01587         template <std::size_t k, class InsertIter>
01588         void up(const std::set<SimplexID<k>&& simplices, InsertIter iter) const
01589         {
01590             for (auto simplex : simplices)
01591             {
01592                 for (auto p : simplex.ptr->_up)
01593                 {
01594                     *iter++ = SimplexID<k+1>(p.second);
01595                 }
01596             }
01597         }
01598 
01599         template <std::size_t k, class InsertIter>
01600         void up(const std::set<SimplexID<k>& simplices, InsertIter iter) const
01601         {
01602             for (auto simplex : simplices)
01603             {
01604                 for (auto p : simplex.ptr->_up)
01605                 {
01606                     *iter++ = SimplexID<k+1>(p.second);
01607                 }
01608             }
01609         }
01610 
01611         template <std::size_t k, class InsertIter>
01612         void up(const SimplexID<k> simplex, InsertIter iter) const
01613         {
01614             for (auto p : simplex.ptr->_up)
01615             {
01616                 *iter++ = SimplexID<k+1>(p.second);
01617             }
01618         }
01619 
01620         /**
01621          * @brief      Get the boundary of a set of simplices.
01622          *
01623          * @param      simplices  The set of simplicies.
01624          *
01625          * @tparam     k      The dimension of the simplices.
01626          *
01627          * @return     The set of boundary simplices.
01628          */
01629         template <std::size_t k>
01630         std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &&simplices) const
01631         {
01632             std::set<SimplexID<k-1> > rval;
01633             for (auto nid : simplices)
01634             {
01635                 for (auto p : nid.ptr->_down)
01636                 {
01637                     rval.insert(SimplexID<k-1>(p.second));
01638                 }
01639             }
01640             return rval;
01641         }
01642 
01643         /**
```

```
01644          * @brief      Get the boundary of a set of simplices.
01645          *
01646          * @param      simplices  The set of simplicies.
01647          *
01648          * @tparam     k      The dimension of the simplices.
01649          *
01650          * @return     The set of boundary simplices.
01651          */
01652         template <std::size_t k>
01653         std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &simplices) const
01654         {
01655             std::set<SimplexID<k-1> > rval;
01656             for (auto simplex : simplices)
01657             {
01658                 for (auto p : simplex.ptr->_down)
01659                 {
01660                     rval.insert(SimplexID<k-1>(p.second));
01661                 }
01662             }
01663             return rval;
01664         }
01665
01666         /**
01667          * @brief      Get the boundary of a simplex.
01668          *
01669          * @param      simplex   The simplex of interest.
01670          *
01671          * @tparam     k     The dimension of the simplex.
01672          *
01673          * @return     Set of (k-1)-simplices of which 'simplex' is a coface of.
01674          */
01675         template <std::size_t k>
01676         std::set<SimplexID<k-1> > down(const SimplexID<k> simplex) const
01677         {
01678             std::set<SimplexID<k-1> > rval;
01679             for (auto p : simplex.ptr->_down)
01680             {
01681                 rval.insert(SimplexID<k-1>(p.second));
01682             }
01683             return rval;
01684         }
01685
01686         template <std::size_t k, class InsertIter>
01687         void down(const std::set<SimplexID<k>&& simplices, InsertIter iter) const{
01688             for (auto simplex : simplices)
01689             {
01690                 for (auto p : simplex.ptr->_down)
01691                 {
01692                     *iter++ = SimplexID<k-1>(p.second);
01693                 }
01694             }
01695         }
01696
01697         template <std::size_t k, class InsertIter>
01698         void down(const std::set<SimplexID<k>& simplices, InsertIter iter) const{
01699             for (auto simplex : simplices)
01700             {
01701                 for (auto p : simplex.ptr->_down)
01702                 {
01703                     *iter++ = SimplexID<k-1>(p.second);
01704                 }
01705             }
01706         }
01707
01708         template <std::size_t k, class InsertIter>
01709         void down(const SimplexID<k> simplex, InsertIter iter) const{
01710             for (auto p : simplex.ptr->_down)
01711             {
01712                 *iter++ = SimplexID<k-1>(p.second);
01713             }
01714         }
01715
01716         /**
01717          * @brief      Gets the edge up from a simplex.
01718          *
01719          * @param[in]  simplex   The simplex of interest.
01720          * @param[in]  a     Key of the edge to get.
01721          *
01722          * @tparam     k     The level of the simplex of interest
01723          *
01724          * @return     The edge up.
01725          */
01726         template <std::size_t k>
01727         EdgeID<k+1> get_edge_up(SimplexID<k> simplex, KeyType a)
01728         {
01729             return EdgeID<k+1>(simplex.ptr->_up.at(a), a);
01730         }
```

```
01731
01732         /**
01733          * @brief      Gets the edge down from a simplex.
01734          *
01735          * @param[in]  simplex   The simplex of interest.
01736          * @param[in]  a     Key of the edge to get.
01737          *
01738          * @tparam     k     The level of the simplex of interest
01739          *
01740          * @return     The edge down.
01741          */
01742         template <std::size_t k>
01743         EdgeID<k> get_edge_down(SimplexID<k> simplex, KeyType a)
01744         {
01745             return EdgeID<k>(simplex.ptr, a);
01746         }
01747
01748         /**
01749          * @brief      Gets the edge up from a simplex.
01750          *
01751          * @param[in]  simplex    The simplex of interest.
01752          * @param[in]  a      Key of the edge to get.
01753          *
01754          * @tparam     k      The level of the simplex of interest
01755          *
01756          * @return     The edge up.
01757          */
01758         template <std::size_t k>
01759         EdgeID<k+1> get_edge_up(SimplexID<k> simplex, KeyType a) const
01760         {
01761             return EdgeID<k+1>(simplex.ptr->_up.at(a), a);
01762         }
01763
01764         /**
01765          * @brief      Gets the edge down from a simplex.
01766          *
01767          * @param[in]  simplex   The simplex of interest.
01768          * @param[in]  a     Key of the edge to get.
01769          *
01770          * @tparam     k     The level of the simplex of interest
01771          *
01772          * @return     The edge down.
01773          */
01774         template <std::size_t k>
01775         EdgeID<k> get_edge_down(SimplexID<k> simplex, KeyType a) const
01776         {
01777             return EdgeID<k>(simplex.ptr, a);
01778         }
01779
01780         /**
01781          * @brief      Check whether a simplex with some name exists.
01782          *
01783          * @param[in]  s     C-style array of the name
01784          *
01785          * @tparam     k     The dimension of the simplex.
01786          *
01787          * @return     True if the simplex is in the complex.
01788          */
01789         template <std::size_t k>
01790         bool exists(const KeyType (&s)[k]) const
01791         {
01792
01793             return get_recurse<0, k>::apply(s, _root) != nullptr;
01794         }
01795
01796         /**
01797          * @brief      Get the number of simplices of dimension 'k'.
01798          *
01799          * @tparam     k     The dimension of interest.
01800          *
01801          * @return     Integer number of k-simplices in the complex.
01802          */
01803         template <std::size_t k>
01804         std::size_t size() const
01805         {
01806             return std::get<k>(levels).size();
01807         }
01808
01809
01810         // template <std::size_t k> using SimplexIDIterator = detail::node_id_iterator<typename
      detail::map<NodePtr<k»::iterator, SimplexID<k»;
01811         /**
01812          * @brief      Create an iterator to traverse the SimplexIDs of a
01813          *             dimension.
01814          *
01815          * @tparam     k     The simplex dimension to traverse.
01816          *
```

```
01817          * @return     An iterator across all k-simplices of the complex.
01818          */
01819         template <std::size_t k>
01820         auto get_level_id()
01821         {
01822             auto begin = std::get<k>(levels).begin();
01823             auto end = std::get<k>(levels).end();
01824             auto data_begin = detail::make_node_id_iterator<decltype(begin), SimplexID<k> >(begin);
01825             auto data_end   = detail::make_node_id_iterator<decltype(end), SimplexID<k> >(end);
01826             return util::make_range(data_begin, data_end);
01827         }
01828
01829         /**
01830          * @brief      Create an iterator to traverse the SimplexIDs of a
01831          *             dimension.
01832          *
01833          * @tparam     k     The simplex dimension to traverse.
01834          *
01835          * @return     An iterator across all k-simplices of the complex.
01836          */
01837         template <std::size_t k>
01838         auto get_level_id() const
01839         {
01840             auto begin = std::get<k>(levels).cbegin();
01841             auto end = std::get<k>(levels).cend();
01842             auto data_begin = detail::make_node_id_iterator<decltype(begin), const SimplexID<k>
      >(begin);
01843             auto data_end   = detail::make_node_id_iterator<decltype(end), const SimplexID<k> >(end);
01844             return util::make_range(data_begin, data_end);
01845         }
01846
01847
01848         // template <std::size_t k> using DataIterator = detail::node_data_iterator<typename
      std::map<std::size_t, NodePtr<k»::iterator, NodeData<k»;
01849         /**
01850          * @brief      Create an iterator to traverse the simplex data of a
01851          *             dimension.
01852          *
01853          * @tparam     k     The simplex dimension to traverse.
01854          *
01855          * @return     An iterator across the data of all k-simplices in the
01856          * complex.
01857          */
01858         template <std::size_t k>
01859         auto get_level()
01860         {
01861             auto begin = std::get<k>(levels).begin();
01862             auto end = std::get<k>(levels).end();
01863             auto data_begin = detail::make_node_data_iterator<decltype(begin), NodeData<k> >(begin);
01864             auto data_end   = detail::make_node_data_iterator<decltype(end), NodeData<k> >(end);
01865             return util::make_range(data_begin, data_end);
01866         }
01867
01868         /**
01869          * @brief      Create an iterator to traverse the simplex data of a
01870          *             dimension.
01871          *
01872          * @tparam     k     The simplex dimension to traverse.
01873          *
01874          * @return     An iterator across the data of all k-simplices in the
01875          * complex.
01876          */
01877         template <std::size_t k>
01878         auto get_level() const
01879         {
01880             auto begin = std::get<k>(levels).cbegin();
01881             auto end = std::get<k>(levels).cend();
01882             auto data_begin = detail::make_node_data_iterator<decltype(begin), const NodeData<k>
      >(begin);
01883             auto data_end   = detail::make_node_data_iterator<decltype(end), const NodeData<k> >(end);
01884             return util::make_range(data_begin, data_end);
01885         }
01886
01887         /**
01888          * @brief      Remove a simplex and all dependent simplices by name.
01889          *
01890          * @param[in]  s     C-style array with the name of the simplex to
01891          *                   remove.
01892          *
01893          * @tparam     k     The dimension of the simplex.
01894          *
01895          * @return     Integer corresponding to the number of simplices removed.
01896          */
01897         template <std::size_t k>
01898         std::size_t remove(const KeyType (&s)[k])
01899         {
01900             Node<k>* root  = get_recurse<0, k>::apply(s, _root);
```

```
01901             std::size_t   count = 0;
01902             return remove_recurse<k, 0>::apply(this, &root, &root + 1, count);
01903         }
01904 
01905         /**
01906          * @brief      Remove a simplex and all dependent simplices by name.
01907          *
01908          * @param[in]  s     std::array with the name of the simplex to remove.
01909          *
01910          * @tparam     k     The dimension of the simplex.
01911          *
01912          * @return     Integer corresponding to the number of simplices removed.
01913          */
01914         template <std::size_t k>
01915         std::size_t remove(const std::array<KeyType, k> &s)
01916         {
01917             Node<k>* root  = get_recurse<0, k>::apply(s.data(), _root);
01918             std::size_t   count = 0;
01919             return remove_recurse<k, 0>::apply(this, &root, &root + 1, count);
01920         }
01921 
01922         /**
01923          * @brief      Remove a simplex and all dependent simplices by
01924          *             SimplexID.
01925          *
01926          * @param[in]  s     SimplexID of the simplex to remove.
01927          *
01928          * @tparam     k     The dimension of the simplex.
01929          *
01930          * @return     Integer corresponding to the number of simplices removed.
01931          */
01932         template <std::size_t k>
01933         std::size_t remove(SimplexID<k> s)
01934         {
01935             std::size_t count = 0;
01936             return remove_recurse<k, 0>::apply(this, &s.ptr, &s.ptr + 1, count);
01937         }
01938 
01939         /**
01940          * @brief      Checks whether a simplex is on a boundary.
01941          *
01942          * @param[in]  s     SimplexID of interest
01943          *
01944          * @tparam     k     Dimension of the simplex
01945          *
01946          * @return     True if the simplex is a member of a topLevel-1 simplex
01947          *             on the boundary or if the simplex is on a boundary or if
01948          *             the simplex is a coboundary of a boundary topLevel-1
01949          *             simplex.
01950          */
01951         template <std::size_t k>
01952         bool onBoundary(const SimplexID<k> s) const
01953         {
01954             return onBoundaryH<k, 0>::apply(s);
01955         }
01956 
01957 
01958         /**
01959          * @brief      Checks whether a simplex is near a boundary.
01960          *
01961          * @param[in]  s      SimplexID of interest
01962          *
01963          * @tparam     level  Dimension of the simplex
01964          *
01965          * @return     True if the simplex or any subsimplices are onBoundary.
01966          */
01967         template <std::size_t level>
01968         bool nearBoundary(const SimplexID<level> s) const
01969         {
01970             return nearBoundaryH<level, 0>::apply(s);
01971         }
01972 
01973         //** Reintroduce this code block when this is resolved
01974         // http://www.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#727
01975 
01976         // /**
01977         //  * @brief      Checks whether a simplex is on a boundary.
01978         //  *
01979         //  * @param[in]  s     SimplexID of interest
01980         //  *
01981         //  * @tparam     k     Dimension of the simplex
01982         //  *
01983         //  * @return     True if the simplex interacts with a
01984         //  *             topLevel-1 simplex which is on a boundary.
01985         //  */
01986         // template <std::size_t k>
01987         // bool onBoundary(const SimplexID<k> s) const
```

```
01988         // {
01989         //     for(auto p : s.ptr->_up)
01990         //     {
01991         //         if(onBoundary(SimplexID<k+1>(p.second)))
01992         //             return true;
01993         //     }
01994         //     return false;
01995         // }
01996
01997         // /**
01998         //  * @brief      Specialization of the facets
01999         //  *
02000         //  * @param[in]  s     SimplexID of interest
02001         //  *
02002         //  * @tparam     k     Dimension of the simplex
02003         //  *
02004         //  * @return     True if s is on a boundary
02005         //  */
02006         // template<>
02007         // bool onBoundary(const SimplexID<topLevel> s) const
02008         // {
02009         //     for(auto p : s.ptr->_down){
02010         //         if(onBoundary(SimplexID<topLevel-1>(p.second)))
02011         //             return true;
02012         //     }
02013         //     return false;
02014         // }
02015
02016         // /**
02017         //  * @brief      Specialization of the topLevel-1 simplices
02018         //  *
02019         //  * @param[in]  s     SimplexID of interest
02020         //  *
02021         //  * @tparam     k     Dimension of the simplex
02022         //  *
02023         //  * @return     True if s is on a boundary
02024         //  */
02025         // template<>
02026         // bool onBoundary(const SimplexID<topLevel-1> s) const
02027         // {
02028         //     return s.ptr->_up.size() != 2;
02029         // }
02030
02031
02032         /**
02033          * @brief      Less than or equal to comparison operator of two
02034          *             SimplexIDs.
02035          *
02036          * @param[in]  lhs   The left hand side
02037          * @param[in]  rhs   The right hand side
02038          *
02039          * @tparam     L     Dimension of lhs simplex.
02040          * @tparam     R     Dimension of rhs simplex.
02041          *
02042          * @return     True if lhs is rhs or a proper face of rhs.
02043          */
02044         template <std::size_t L, std::size_t R>
02045         bool leq(SimplexID<L> lhs, SimplexID<R> rhs) const
02046         {
02047             auto        name_lhs = get_name(lhs);
02048             auto        name_rhs = get_name(rhs);
02049
02050             std::size_t i = 0;
02051             for (std::size_t j = 0; i < L && j < R; ++j)
02052             {
02053                 if (name_lhs[i] == name_rhs[j])
02054                 {
02055                     ++i;
02056                 }
02057             }
02058             return (i == L);
02059         }
02060
02061         /**
02062          * @brief      Equality comparison of two simplices.
02063          *
02064          * @param[in]  lhs   The left hand side
02065          * @param[in]  rhs   The right hand side
02066          *
02067          * @tparam     L     Dimension of lhs simplex.
02068          * @tparam     R     Dimension of rhs simplex.
02069          *
02070          * @return     Always false as L != R. The L==R case is overloaded by
02071          *             partial specialization.
02072          */
02073         template <std::size_t L, std::size_t R>
02074         bool eq(SimplexID<L>, SimplexID<R>) const
```

```
02075         {
02076             return false;
02077         }
02078 
02079         /**
02080          * @brief      Equality comparison of two simplices.
02081          *
02082          * @param[in]  lhs   The left hand side
02083          * @param[in]  rhs   The right hand side
02084          *
02085          * @tparam     k     Dimension of the simplices.
02086          *
02087          * @return     True if the names are the same.
02088          */
02089         template <std::size_t k>
02090         bool eq(SimplexID<k> lhs, SimplexID<k> rhs) const
02091         {
02092             auto name_lhs = get_name(lhs);
02093             auto name_rhs = get_name(rhs);
02094 
02095             for (std::size_t i = 0; i < k; ++i)
02096             {
02097                 if (name_lhs[i] != name_rhs[i])
02098                 {
02099                     return false;
02100                 }
02101             }
02102             return true;
02103         }
02104 
02105         /**
02106          * @brief      Less than comparison of simplices.
02107          *
02108          * @param[in]  lhs   The left hand side
02109          * @param[in]  rhs   The right hand side
02110          *
02111          * @tparam     L     Dimension of lhs simplex.
02112          * @tparam     R     Dimension of rhs simplex.
02113          *
02114          * @return     True if lhs is a proper subface of rhs.
02115          */
02116         template <std::size_t L, std::size_t R>
02117         bool lt(SimplexID<L> lhs, SimplexID<R> rhs) const
02118         {
02119             return L < R && leq(lhs, rhs);
02120         }
02121 
02122     private:
02123         /**
02124          * @brief      Base case for checking if simplex is near a boundary
02125          *
02126          * @tparam     level  Dimension of the simplex
02127          * @tparam     foo    Dummy argument to avoid explicit specialization in
02128          *                    class scope
02129          */
02130         template <std::size_t level, std::size_t foo>
02131         struct nearBoundaryH
02132         {
02133             static bool apply(const SimplexID<level> s){
02134                 auto name = s.indices();
02135                 KeyType down[level-1];
02136 
02137                 for(std::size_t i = 0; i < level; ++i){
02138                     std::size_t k = 0;
02139                     for(std::size_t j = 0; j < level; ++j){
02140                         if (i != j){
02141                             down[k++] = name[j];
02142                         }
02143                     }
02144                     if(onBoundaryH<1, 0>::apply(
02145                         get_down_recurse<level, level-1>::apply(down, s.ptr)
02146                     ))
02147                         return true;
02148                 }
02149                 return false;
02150             }
02151         };
02152 
02153         /**
02154          * @brief      Specialization of vertices
02155          *
02156          * @tparam     foo    Dummy argument to avoid explicit specialization in
02157          *                    class scope
02158          */
02159         template <std::size_t foo>
02160         struct nearBoundaryH<1, foo>
02161         {
```

```
02162             static bool apply(const SimplexID<1> s){
02163                 if(onBoundaryH<1, 0>::apply(s))
02164                     return true;
02165                 return false;
02166             }
02167         };
02168 
02169         /**
02170          * @brief      Base case for checking if simplex is on a boundary
02171          *
02172          * @tparam     level  Dimension of the simplex
02173          * @tparam     foo    Dummy argument to avoid explicit specialization in
02174          *                    class scope
02175          */
02176         template <std::size_t level, std::size_t foo>
02177         struct onBoundaryH
02178         {
02179             /**
02180              * @brief      Recurse up complex to find boundary.
02181              *
02182              * @param[in]  s     Simplex of interest
02183              *
02184              * @return     True if on boundary
02185              */
02186             static bool apply(const SimplexID<level> s)
02187             {
02188                 for(auto p : s.ptr->_up)
02189                 {
02190                     if(onBoundaryH<level+1, foo>::apply(SimplexID<level+1>(p.second)))
02191                         return true;
02192                 }
02193                 return false;
02194             }
02195         };
02196 
02197         /**
02198          * @brief      Specialization for if facets are on boundary.
02199          *
02200          * @tparam     foo   Dummy argument to avoid explicit specialization in
02201          *                   class scope}
02202          */
02203         template <std::size_t foo>
02204         struct onBoundaryH<topLevel, foo>
02205         {
02206             /**
02207              * @brief      Check if a face is on a boundary
02208              *
02209              * @param[in]  s     SimplexID<topLevel> of interest
02210              *
02211              * @return     True if a member SimplexID<topLevel-1> is a boundary.
02212              */
02213             static bool apply(const SimplexID<topLevel> s)
02214             {
02215                 for(auto p : s.ptr->_down){
02216                     if(onBoundaryH<topLevel-1, foo>::apply(SimplexID<topLevel-1>(p.second)))
02217                         return true;
02218                 }
02219                 return false;
02220             }
02221         };
02222 
02223         /**
02224          * @brief      Specialization for topLevel-1 simplices
02225          *
02226          * @tparam     foo   Dummy argument to avoid explicit specialization in
02227          *                   class scope}
02228          */
02229         template <std::size_t foo>
02230         struct onBoundaryH<bdryLevel, foo>
02231         {
02232             /**
02233              * @brief      Check if SimplexID<topLevel-1> is on a boundary
02234              *
02235              * @param[in]  s     SimplexID of interest
02236              *
02237              * @return     True if simplex has less than 2 coboundary faces.
02238              */
02239             static bool apply(const SimplexID<bdryLevel> s)
02240             {
02241                 return s.ptr->_up.size() < 2;
02242             }
02243         };
02244 
02245         /**
02246          * @brief      Base case for recursively deleting simplices.
02247          *
02248          * @tparam     level  Simplex dimension to operate at.
```

```
02249          * @tparam     foo    Dummy argument to avoid explicit specialization in
02250          *                    class scope
02251          */
02252         template <std::size_t level, std::size_t foo>
02253         struct remove_recurse
02254         {
02255             /**
02256              * @brief      Recursively remove simplices.
02257              *
02258              * @param      that   The CASC object
02259              * @param[in]  begin  Iterator to beginning of the set of simplices
02260              *                    to remove.
02261              * @param[in]  end    Iterator to the end of the set.
02262              * @param      count  Number of simplices removed already.
02263              *
02264              * @tparam     T      Typename of the iterator.
02265              *
02266              * @return     Recurse to the next level and remove coboundary
02267              *             simplices.
02268              */
02269             template <typename T>
02270             static std::size_t apply(type_this* that, T begin, T end, std::size_t &count)
02271             {
02272                 std::set<Node<level+1>*> next;
02273                 // for each node of interest...
02274                 for (auto i = begin; i != end; ++i)
02275                 {
02276                     auto up = (*i)->_up;
02277                     for (auto j = up.begin(); j != up.end(); ++j)
02278                     {
02279                         next.insert(j->second);
02280                     }
02281                     that->remove_node(*i);
02282                     ++count;
02283                 }
02284                 return remove_recurse<level+1, foo>::apply(that, next.begin(), next.end(), count);
02285             }
02286         };
02287
02288         /**
02289          * @brief      Terminal condition for remove_recurse.
02290          *
02291          * @tparam     foo   Dummy argument to avoid explicit specialization in
02292          *                   class scope
02293          */
02294         template <std::size_t foo>
02295         struct remove_recurse<topLevel, foo>
02296         {
02297             /**
02298              * @brief      Remove the facets of the complex.
02299              *
02300              * @param      that   The CASC object
02301              * @param[in]  begin  Iterator to beginning of the set of simplices
02302              *                    to remove.
02303              * @param[in]  end    Iterator to the end of the set.
02304              * @param      count  Number of simplices removed already.
02305              *
02306              * @tparam     T      Typename of the iterator.
02307              *
02308              * @return     The number of simplices removed
02309              */
02310             template <typename T>
02311             static std::size_t apply(type_this* that, T begin, T end, std::size_t &count)
02312             {
02313                 for (auto i = begin; i != end; ++i)
02314                 {
02315                     that->remove_node(*i);
02316                     ++count;
02317                 }
02318                 return count;
02319             }
02320         };
02321
02322         /**
02323          * @brief      Recursively retrieve a simplex of interest.
02324          *
02325          * @tparam     level  The current simplex dimension.
02326          * @tparam     n      Number of remaining times to recurse.
02327          */
02328         template <std::size_t level, std::size_t n>
02329         struct get_recurse
02330         {
02331             /**
02332              * @brief      Get the simplex of interest.
02333              *
02334              * @param[in]  that  The simplicial complex to search.
02335              * @param[in]  s     Pointer to an array of Keys.
```

```
02336              * @param      root  The current simplex
02337              *
02338              * @return     Returns a pointer to the node.
02339              */
02340             static Node<level+n>* apply(const KeyType* s, Node<level>* root)
02341             {
02342                 // TODO: We probably don't need to check if root is a valid
02343                 // simplex (10)
02344                 if (root)
02345                 {
02346                     auto p = root->_up.find(*s);
02347                     if (p != root->_up.end())
02348                     {
02349                         return get_recurse<level+1, n-1>::apply(s+1, root->_up.at(*s));
02350                     }
02351                     else
02352                     {
02353                         return nullptr;
02354                     }
02355                 }
02356                 else
02357                 {
02358                     return nullptr;
02359                 }
02360             }
02361         };
02362         /**
02363          * @brief      Recursively retrieve a simplex of interest.
02364          *
02365          * @tparam     level  The current simplex dimension.
02366          */
02367         template <std::size_t level>
02368         struct  get_recurse<level, 0>
02369         {
02370             /**
02371              * @brief      Get the simplex of interest.
02372              *
02373              * @param[in]  that  The simplicial complex to search.
02374              * @param[in]  s     Pointer to an array of Keys.
02375              * @param      root  The current simplex
02376              *
02377              * @return     Returns a pointer to the node.
02378              */
02379             static Node<level>* apply(const KeyType*, Node<level>* root)
02380             {
02381                 return root;
02382             }
02383         };
02384
02385         /**
02386          * @brief      Recursively retrieve a simplex of interest going down.
02387          *
02388          * @tparam     level  The current simplex dimension.
02389          * @tparam     n      Number of remaining times to recurse.
02390          */
02391         template <std::size_t level, std::size_t n>
02392         struct get_down_recurse
02393         {
02394             /**
02395              * @brief      Get the simplex of interest.
02396              *
02397              * @param[in]  that  The simplicial complex to search.
02398              * @param[in]  s     Pointer to an array of Keys.
02399              * @param      root  The current simplex
02400              *
02401              * @return     Returns a pointer to the node.
02402              */
02403             static Node<level-n>* apply(const KeyType* s, Node<level>* root)
02404             {
02405                 if (root)
02406                 {
02407                     auto p = root->_down.find(*s);
02408                     if (p != root->_down.end())
02409                     {
02410                         return get_down_recurse<level-1, n-1>::apply(s+1, root->_down[*s]);
02411                     }
02412                     else
02413                     {
02414                         return nullptr;
02415                     }
02416                 }
02417                 else
02418                 {
02419                     return nullptr;
02420                 }
02421             }
02422         };
```

```
02423
02424         /**
02425          * @brief      Recursively retrieve a simplex of interest going down.
02426          *
02427          * @tparam     level  The current simplex dimension.
02428          */
02429         template <std::size_t level>
02430         struct  get_down_recurse<level, 0>
02431         {
02432             /**
02433              * @brief      Get the simplex of interest.
02434              *
02435              * @param[in]  this  The simplicial complex to search.
02436              * @param[in]  s     Pointer to an array of Keys.
02437              * @param      root  The current simplex
02438              *
02439              * @return     Returns a pointer to the node.
02440              */
02441             static Node<level>* apply(const KeyType*, Node<level>* root)
02442             {
02443                 return root;
02444             }
02445         };
02446
02447         /**
02448          * @brief      Insert a simplex and all dependent simplices into the
02449          *             complex.
02450          *
02451          * @tparam     level  Dimension of the current root simplex
02452          * @tparam     n      The number of times to recurse.
02453          */
02454         template <std::size_t level, std::size_t n>
02455         struct insert_full
02456         {
02457             /**
02458              * @brief      Kick off a for loop to insert all cofaces.
02459              *
02460              * @param      that   The simplicial complex
02461              * @param      root   The current simplex to insert at.
02462              * @param[in]  begin  Pointer to an array of Keys.
02463              *
02464              * @return     Returns the node to insert.
02465              */
02466             static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
02467             {
02468                 return insert_for<level, n, n>::apply(that, root, begin);
02469             }
02470         };
02471
02472
02473         /**
02474          * @brief      Insert a simplex and all dependent simplices into the
02475          *             complex.
02476          *
02477          * @tparam     level  Dimension of the current root simplex
02478          */
02479         template <std::size_t level>
02480         struct insert_full<level, 0>
02481         {
02482             /**
02483              * @brief      Terminal case.
02484              *
02485              * @param      that   The simplicial complex
02486              * @param      root   The current simplex to insert at.
02487              * @param[in]  begin  Pointer to an array of Keys.
02488              *
02489              * @return     Returns the node to insert.
02490              */
02491             static Node<level>* apply(type_this*, Node<level>* root, const KeyType*)
02492             {
02493                 return root;
02494             }
02495         };
02496
02497         /**
02498          * @brief      Iterate over antistep
02499          *
02500          * @tparam     level     Dimension of the current root simplex
02501          * @tparam     antistep  Antistep to track which indices to append to root.
02502          * @tparam     n         Original antistep.
02503          */
02504         template <std::size_t level, std::size_t antistep, std::size_t n>
02505         struct insert_for
02506         {
02507             /**
02508              * @brief      Call insert_raw and continue for loop
02509              *
```

```
02510              * @param      that   The simplicial complex
02511              * @param      root   The current simplex to insert at.
02512              * @param[in]  begin  Pointer to an array of Keys.
02513              *
02514              * @return     Returns the node to insert.
02515              */
02516             static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
02517             {
02518                 insert_raw<level, n-antistep>::apply(that, root, begin);
02519                 return insert_for<level, antistep-1, n>::apply(that, root, begin);
02520             }
02521         };
02522
02523         /**
02524          * @brief      Terminal case.
02525          *
02526          * @tparam     level  Dimension of the current root simplex.
02527          * @tparam     n      Original antistep.
02528          */
02529         template <std::size_t level, std::size_t n>
02530         struct insert_for<level, 1, n>
02531         {
02532             /**
02533              * @brief      Call insert_raw and stop loop
02534              *
02535              * @param      that   The simplicial complex
02536              * @param      root   The current simplex to insert at.
02537              * @param[in]  begin  Pointer to an array of Keys.
02538              *
02539              * @return     Returns the node to insert.
02540              */
02541             static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
02542             {
02543                 return insert_raw<level, n-1>::apply(that, root, begin);
02544             }
02545         };
02546
02547         /**
02548          * @brief      Actually insert the node and connect up and down.
02549          *
02550          * @tparam     level  Dimension of the current root simplex.
02551          * @tparam     n      The index to append to root.
02552          */
02553         template <std::size_t level, std::size_t n>
02554         struct insert_raw
02555         {
02556             /**
02557              * @brief      Create the node and connect up and down.
02558              *
02559              * @param      that   The simplicial complex
02560              * @param      root   The current simplex to insert at.
02561              * @param[in]  begin  Pointer to an array of Keys.
02562              *
02563              * @return     Returns the node to insert.
02564              */
02565             static Node<level+n+1>* apply(type_this* that, Node<level>* root, const KeyType* begin)
02566             {
02567
02568                 KeyType        v = *(begin+n);
02569                 Node<level+1>* nn;
02570                 // if root->v doesn't exist then create it
02571                 auto           iter = root->_up.find(v);
02572                 if (iter == root->_up.end())
02573                 {
02574                     nn = that->create_node<level+1>();
02575
02576                     nn->_down[v] = root;
02577                     root->_up[v] = nn;
02578                     that->backfill(root, nn, v);
02579                 }
02580                 else
02581                 {
02582                     nn = iter->second; // otherwise get it
02583                 }
02584                 return insert_full<level+1, n>::apply(that, nn, begin);
02585             }
02586         };
02587
02588         /**
02589          * @brief      Backfill in the pointers from prior nodes to the new node
02590          *
02591          * @param      root   is a parent node
02592          * @param      nn     is the new child node
02593          * @param      value  is the exposed id of nn
02594          * @return     void
02595          *
02596          * @tparam     level  Dimension of the current root simplex.
```

```
02597          */
02598         template <std::size_t level>
02599         void backfill(Node<level>* root, Node<level+1>* nn, KeyType value)
02600         {
02601             for (auto curr = root->_down.begin(); curr != root->_down.end(); ++curr)
02602             {
02603                 int            v = curr->first;
02604 
02605                 Node<level-1>* parent = curr->second;
02606                 Node<level>  * child  = parent->_up[value];
02607 
02608                 nn->_down[v]  = child;
02609                 child->_up[v] = nn;
02610             }
02611         }
02612 
02613         /**
02614          * @brief      Fill in the pointers from level 1 to 0.
02615          *
02616          * @param      root   is a level 0 node
02617          * @param      nn     is a level 1 node
02618          * @param      value  is the exposed id of nn
02619          * @return     void
02620          */
02621         void backfill(Node<0>*, Node<1>*, int)
02622         {
02623             return;
02624         }
02625 
02626         /**
02627          * @brief      Creates a new node of some dimension.
02628          *
02629          * @param[in]  x      Argument to help deduce the new node dimension
02630          *
02631          * @tparam     level  Simplex dimension
02632          *
02633          * @return     A pointer to the new node.
02634          */
02635         template <std::size_t level>
02636         Node<level>* create_node()
02637         {
02638             // Create the new node
02639             auto p = new Node<level>(node_count++);
02640             ++(level_count[level]); // Increment the count in the level
02641 
02642             // node_count-1 to match the internal IDs correctly.
02643             MAYBE_UNUSED bool ret = std::get<level>(levels).insert(
02644                     std::pair<std::size_t, NodePtr<level> >(node_count-1, p)).second;
02645             assert(ret);
02646             /*
02647                // sanity check to make sure there aren't duplicate keys...
02648                if (ret==false) {
02649                 std::cout « "Error: Node '" « node_count « "' already existed
02650                    with value " « *p « std::endl;
02651                }
02652              */
02653             return p;
02654         }
02655 
02656         /**
02657          * @brief      Removes a node.
02658          *
02659          * @param      p      Simplex to remove
02660          *
02661          * @tparam     level  Dimension of the simplex
02662          */
02663         template <std::size_t level>
02664         void remove_node(Node<level>* p)
02665         {
02666             for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
02667             {
02668                 curr->second->_up.erase(curr->first);
02669             }
02670             for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
02671             {
02672                 curr->second->_down.erase(curr->first);
02673             }
02674             --(level_count[level]);
02675             std::get<level>(levels).erase(p->_node);
02676             delete p;
02677         }
02678 
02679         /**
02680          * @brief      Removes a node.
02681          *
02682          * @param      p     Simplex to remove
02683          */
```

```
02684         void remove_node(Node<1>* p)
02685         {
02686             // This for loop should only have a single iteration.
02687             for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
02688             {
02689                 unused_vertices.insert(curr->first);
02690                 curr->second->_up.erase(curr->first);
02691             }
02692             for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
02693             {
02694                 curr->second->_down.erase(curr->first);
02695             }
02696             --(level_count[1]);
02697             std::get<1>(levels).erase(p->_node);
02698             delete p;
02699         }
02700 
02701         /**
02702          * @brief      Removes a node.
02703          *
02704          * @param      p     Simplex to remove
02705          */
02706         void remove_node(Node<0>* p)
02707         {
02708             for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
02709             {
02710 
02711                 curr->second->_down.erase(curr->first);
02712             }
02713             --(level_count[0]);
02714             std::get<0>(levels).erase(p->_node);
02715             delete p;
02716         }
02717 
02718         /**
02719          * @brief      Removes a node.
02720          *
02721          * @param      p     Simplex to remove
02722          */
02723         void remove_node(Node<topLevel>* p)
02724         {
02725             for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
02726             {
02727                 curr->second->_up.erase(curr->first);
02728             }
02729             --(level_count[topLevel]);
02730             std::get<topLevel>(levels).erase(p->_node);
02731             delete p;
02732         }
02733 
02734         /// The root node
02735         NodePtr<0> _root;
02736         /// A counter of the total number of nodes.
02737         std::size_t node_count;
02738         /// A counter of the number of simplices per level.
02739         std::array<std::size_t, numLevels> level_count;
02740         /// Typename of a tuple of LevelIndex broadcasted with NodePtr<k>.
02741         using NodePtrLevel = typename util::int_type_map<std::size_t, std::tuple, LevelIndex,
      NodePtr>::type;
02742         /// Typename of a map of levels to NodePtr<k>*'s.
02743         typename util::type_map<NodePtrLevel, detail::map>::type levels;
02744         /// B-tree of unused vertex indices.
02745         index_tracker::index_tracker<KeyType> unused_vertices;
02746 };
02747 
02748 
02749 /**
02750  * Alias to generate a CASC from a list of traits.
02751  * See also simplicial_complex_traits_default. Example -- To create a
02752  * tetrahedral mesh with integer data on all simplices:
02753  * ~~~~~~~~~~~~~~~{.cpp}
02754  * auto mesh = AbstractSimplicialComplex<
02755  *     int, // KEYTYPE
02756  *     int, // Root data
02757  *     int, // Vertex data
02758  *     int, // Edge data
02759  *     int, // Face data
02760  *     int  // Volume data
02761  * >();
02762  * ~~~~~~~~~~~~~~~
02763  */
02764 template <typename KeyType, typename ... Ts>
02765 using AbstractSimplicialComplex = simplicial_complex<
02766             detail::simplicial_complex_traits_default<KeyType, Ts...> >;
02767 
02768 /// @cond detail
02769 namespace simplex_set_detail{
```

```
02770 /**
02771  * @brief      Template to compute a hash for a SimplexID.
02772  *
02773  * Since SimplexID is actually a wrapper around a Node* we have to hash it
02774  * accordingly. The static_cast calls the defined explicit operator which
02775  * reinterprets the stored Node* pointer as a uintptr_t which we can hash
02776  * directly.
02777  *
02778  * @tparam     SimplexID  Typename of the SimplexID.
02779  */
02780 template <typename SimplexID>
02781 struct hashSimplexID{
02782     /**
02783      * @brief   Compute the hash.
02784      *
02785      * ~~~~~~~~~~~~~~~~~(.cpp)
02786      * std::cout << hashSimplexID<decltype(nid)>{}(nid) << std::endl;
02787      * ~~~~~~~~~~~~~~~~~
02788      *
02789      * @param[in]  nid   The simplex of interest.
02790      * @return     Resultant hash.
02791      */
02792     std::size_t operator()(const SimplexID nid) const
02793     {
02794         return std::hash<std::uintptr_t>()(static_cast<uintptr_t>(nid));
02795     }
02796 };
02797 } // end namespace simplex_set_detail
02798 /// @endcond
02799
02800 /// Helpful alias defining a unordered_set of simplices. See also hashSimplexID.
02801 template <typename T> using NodeSet =
02802         std::unordered_set<T, simplex_set_detail::hashSimplexID<T> >;
02803 } // end namespace casc
```

10.17 include/casc/stringutil.h File Reference

String utilities for CASC.

```
#include <string>
```

Namespaces

- namespace casc

  Namespace for everything CASC.

Functions

- template<typename T , std::size_t k>
  std::string casc::to_string (const std::array< T, k > &A)

  Returns a string representation of the vertex subsimplicies of a given simplex.

10.18 stringutil.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
```

```
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ****************************************************************************
00023  */
00024
00025 /**
00026  * @file  stringutil.h
00027  * @brief String utilities for CASC.
00028  */
00029
00030 #pragma once
00031
00032 #include <string>
00033
00034 namespace casc
00035 {
00036 /**
00037  * @brief      Returns a string representation of the vertex subsimplicies
00038  *             of a given simplex
00039  *
00040  * @param[in]  A     Array containing name of a simplex.
00041  *
00042  * @tparam     T     Typename KeyType.
00043  * @tparam     k     Dimension of the simplex.
00044  *
00045  * @return     String representation of the object.
00046  */
00047 template <typename T, std::size_t k>
00048 std::string to_string(const std::array<T,k>& A)
00049 {
00050     if (k==0){
00051         return "{root}";
00052     }
00053     std::string out;
00054     out += "{";
00055     for(int i = 0; i + 1 < k; ++i)
00056     {
00057         out += std::to_string(A[i]) + ",";
00058     }
00059     if(k > 0)
00060     {
00061         out += std::to_string(A[k-1]);
00062     }
00063     out += "}";
00064     return out;
00065 }
00066 } // end namespace casc
```

10.19 include/casc/typetraits.h File Reference

Helper functions for debugging template types.

Functions

- template<class T >
  CONSTEXPR14_TN static_string type_name ()

  Print the typename of an object at compile time.

10.19.1 Detailed Description

This is copied directly from this very helpful post from Stackoverflow.

10.19.2 Function Documentation

10.19.2.1 type_name()

```
template<class T >
CONSTEXPR14_TN static_string type_name ( )
```

Example usage:
```
std::cout « "decltype(i) is " « type_name<decltype(i)>() « '\n';
```

10.20 typetraits.h

Go to the documentation of this file.

```
00001 /**
00002  * @file  typetraits.h
00003  * @brief Helper functions for debugging template types.
00004  *
00005  * This is copied directly from this very helpful post from
00006  * <a
      href="https://stackoverflow.com/questions/81870/is-it-possible-to-print-a-variables-type-in-standard-c/20170989#20170989
00007  */
00008
00009 /// @cond hidden
00010 #pragma once
00011
00012 #include <cstddef>
00013 #include <stdexcept>
00014 #include <cstring>
00015 #include <ostream>
00016
00017 #ifndef _MSC_VER
00018 #  if __cplusplus < 201103
00019 #    define CONSTEXPR11_TN
00020 #    define CONSTEXPR14_TN
00021 #    define NOEXCEPT_TN
00022 #  elif __cplusplus < 201402
00023 #    define CONSTEXPR11_TN constexpr
00024 #    define CONSTEXPR14_TN
00025 #    define NOEXCEPT_TN noexcept
00026 #  else
00027 #    define CONSTEXPR11_TN constexpr
00028 #    define CONSTEXPR14_TN constexpr
00029 #    define NOEXCEPT_TN noexcept
00030 #  endif
00031 #else  // _MSC_VER
00032 #  if _MSC_VER < 1900
00033 #    define CONSTEXPR11_TN
00034 #    define CONSTEXPR14_TN
00035 #    define NOEXCEPT_TN
00036 #  elif _MSC_VER < 2000
00037 #    define CONSTEXPR11_TN constexpr
00038 #    define CONSTEXPR14_TN
00039 #    define NOEXCEPT_TN noexcept
00040 #  else
00041 #    define CONSTEXPR11_TN constexpr
00042 #    define CONSTEXPR14_TN constexpr
00043 #    define NOEXCEPT_TN noexcept
00044 #  endif
00045 #endif  // _MSC_VER
00046
00047 class static_string
00048 {
00049     const char* const p_;
00050     const std::size_t sz_;
00051
00052 public:
00053     typedef const char* const_iterator;
00054
00055     template <std::size_t N>
00056     CONSTEXPR11_TN static_string(const char(&a)[N]) NOEXCEPT_TN
00057         : p_(a)
```

```
00058         , sz_(N-1)
00059         {}
00060
00061     CONSTEXPR11_TN static_string(const char* p, std::size_t N) NOEXCEPT_TN
00062         : p_(p)
00063         , sz_(N)
00064         {}
00065
00066     CONSTEXPR11_TN const char* data() const NOEXCEPT_TN {return p_;}
00067     CONSTEXPR11_TN std::size_t size() const NOEXCEPT_TN {return sz_;}
00068
00069     CONSTEXPR11_TN const_iterator begin() const NOEXCEPT_TN {return p_;}
00070     CONSTEXPR11_TN const_iterator end()   const NOEXCEPT_TN {return p_ + sz_;}
00071
00072     CONSTEXPR11_TN char operator[](std::size_t n) const
00073     {
00074         return n < sz_ ? p_[n] : throw std::out_of_range("static_string");
00075     }
00076 };
00077
00078 inline
00079 std::ostream&
00080 operator<<(std::ostream& os, static_string const& s)
00081 {
00082     return os.write(s.data(), s.size());
00083 }
00084 /// @endcond
00085
00086 /**
00087  * @brief      Print the typename of an object at compile time.
00088  *
00089  * Example usage:
00090  * ~~~~~~~~~~~~~~~~{.cpp}
00091  * std::cout << "decltype(i) is " << type_name<decltype(i)>() << '\n';
00092  * ~~~~~~~~~~~~~~~~
00093  */
00094 template <class T>
00095 CONSTEXPR14_TN
00096 static_string
00097 type_name()
00098 {
00099 #ifdef __clang__
00100     static_string p = __PRETTY_FUNCTION__;
00101     return static_string(p.data() + 31, p.size() - 31 - 1);
00102 #elif defined(__GNUC__)
00103     static_string p = __PRETTY_FUNCTION__;
00104 #  if __cplusplus < 201402
00105     return static_string(p.data() + 36, p.size() - 36 - 1);
00106 #  else
00107     return static_string(p.data() + 46, p.size() - 46 - 1);
00108 #  endif
00109 #elif defined(_MSC_VER)
00110     static_string p = __FUNCSIG__;
00111     return static_string(p.data() + 38, p.size() - 38 - 7);
00112 #endif
00113 }
```

10.21 include/casc/util.h File Reference

Metatemplate pack expansion helpers.

```
#include <utility>
#include <array>
```

Data Structures

- struct util::range< T >

  A range object to support range based for loops.
- struct util::type_holder< Ts >

  Queue based data structure to hold list of types.
- struct util::type_holder< T, Ts... >

Partial specialization to allow FIFO access of typenames.

- struct util::type_get< k, T >

  Helper to get the kth element from a type_holder.
- struct util::type_get< 0, type_holder< Ts... > >

  Specialization for terminal case.
- struct util::type_get< k, type_holder< Ts... > >

  Specialization to recursively pop types to get the kth type.
- struct util::type_map< C, V >

  Map the types in C into `V<T>`.
- struct util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >

  Maps an integer sequence and typename, F, into outholder.
- struct util::type_swap< TUPLE, HOLDER_FULL >

  Move a list of types from one container to another.
- struct util::type_swap< TUPLE, HOLDER< Ts... > >

  Move a list of types from one container to another.
- struct util::reverse_sequence< Integer, IntegerSequence >

  Reverse an Integer Sequence.
- struct util::remove_first_val< Integer, IntegerSequence >

  General template for removing the first value from a type holder.
- struct util::remove_first_val< Integer, InHolder< Integer, I, Is... > >

  Specialization for removing first integer from a sequence of compile time integers.

Namespaces

- namespace util

  Metatemplate programming utilities namespace.

Functions

- template<typename T >
  range< T > util::make_range (T b, T e)

  Make a range object.
- template<typename T >
  range< T > util::make_range (std::pair< T, T > p)

  Makes a range object.
- template<class Integer , typename IntegerSequence , typename Fn , typename ... Args>
  void util::int_for_each (Fn &&f, Args &&... args)

  Calls a function `f.apply<k>()` for a sequence of integer k's.

10.22 util.h

Go to the documentation of this file.

```
00001 /*
00002  * ******************************************************************************
00003  * This file is part of the Colored Abstract Simplicial Complex library.
00004  * Copyright (C) 2016-2017
00005  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
00006  *    and Michael Holst
00007  *
00008  * This library is free software; you can redistribute it and/or
00009  * modify it under the terms of the GNU Lesser General Public
00010  * License as published by the Free Software Foundation; either
```

```
00011  * version 2.1 of the License, or (at your option) any later version.
00012  *
00013  * This library is distributed in the hope that it will be useful,
00014  * but WITHOUT ANY WARRANTY; without even the implied warranty of
00015  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
00016  * Lesser General Public License for more details.
00017  *
00018  * You should have received a copy of the GNU Lesser General Public
00019  * License along with this library; if not, write to the Free Software
00020  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
00021  *
00022  * ********************************************************************
00023  */
00024
00025 /**
00026  * @file  util.h
00027  * @brief Metatemplate pack expansion helpers
00028  */
00029
00030 #pragma once
00031
00032 #include <utility>
00033 #include <array>
00034
00035 /// Metatemplate programming utilities namespace
00036 namespace util
00037 {
00038 /**
00039  * @brief      A range object to support range based for loops.
00040  *
00041  * This is a basic data structure which implements a `begin()` and `end()`
00042  * functions for range based for lopping added in C++11.
00043  * See also
00044  * <a href="http://en.cppreference.com/w/cpp/language/range-for">range-for</a>.
00045  *
00046  * @tparam     T     Typename of the iterator
00047  */
00048 template<typename T> struct range
00049 {
00050
00051     /**
00052      * @brief      Construct a range for a container class.
00053      *
00054      * @param[in]  c  Container class which implements begin() and end().
00055      *
00056      * @tparam     C  Typename of the container.
00057      */
00058     template <class C>
00059     range(C &&c) : _begin(c.begin()), _end(c.end()) {}
00060
00061     /**
00062      * @brief      Construct a range from an iterator.
00063      *
00064      * @param[in]  b     Beginning iterator
00065      * @param[in]  e     End iterator.
00066      */
00067     range(T b, T e) : _begin(b), _end(e) {}
00068
00069     /**
00070      * @brief      Get the beginning iterator.
00071      *
00072      * @return     Returns an iterator to the beginning.
00073      */
00074     T begin() { return _begin; }
00075
00076     /**
00077      * @brief      Get the end iterator.
00078      *
00079      * @return     Returns an iterator to the end.
00080      */
00081     T end() { return _end; }
00082
00083     private:
00084         /// Iterator to the beginning.
00085         T _begin;
00086         /// Iterator to the end.
00087         T _end;
00088 };
00089
00090 /**
00091  * @brief      Make a range object.
00092  *
00093  * @param[in]  b     Iterator to the beginning.
00094  * @param[in]  e     Iterator to the end.
00095  *
00096  * @tparam     T     Typename of the iterator.
00097  *
```

```
00098  * @return     Returns a range of the iterators.
00099  */
00100 template<typename T> range<T> make_range(T b, T e)
00101 {
00102     return range<T>(std::move(b), std::move(e));
00103 }
00104
00105 /**
00106  * @brief      Makes a range object.
00107  *
00108  * @param[in]  p     A pair containing begin and end iterators.
00109  *
00110  * @tparam     T     Typename of the iterator.
00111  *
00112  * @return     Returns a range of the iterators.
00113  */
00114 template<typename T> range<T> make_range(std::pair<T, T> p)
00115 {
00116     return range<T>(std::move(p.first), std::move(p.second));
00117 }
00118
00119 /**
00120  * @brief      Queue based data structure to hold list of types.
00121  *
00122  * Types in the type_holder can be accessed by accessing the `head` type.
00123  * Subsequent types are in the `tail`. See also type_get.
00124  *
00125  * @tparam     Ts    List of typenames
00126  */
00127 template <typename ... Ts>
00128 struct type_holder
00129 {
00130     /// Length of the list of types
00131     static const std::size_t size = sizeof ... (Ts);
00132 };
00133
00134 /**
00135  * @brief      Partial specialization to allow FIFO access of typenames.
00136  *
00137  * @tparam     T     The first typename
00138  * @tparam     Ts    The following typenames
00139  */
00140 template <typename T, typename ... Ts>
00141 struct type_holder<T, Ts...>
00142 {
00143     /// The first type
00144     using head = T;
00145     /// The following types
00146     using tail = type_holder<Ts...>;
00147     /// Length of the list of types
00148     static const std::size_t size = 1 + type_holder<Ts...>::size;
00149 };
00150
00151 /**
00152  * @brief      Helper to get the kth element from a type_holder.
00153  *
00154  * This is the empty general template which will be later specialized.
00155  *
00156  * @tparam     k     Integer index of the type to retrieve
00157  * @tparam     T     A type_holder queue of typenames
00158  */
00159 template <std::size_t k, typename T>
00160 struct type_get {};
00161
00162 /**
00163  * @brief      Specialization for terminal case.
00164  *
00165  * @tparam     Ts    Following typenames
00166  */
00167 template <typename ... Ts>
00168 struct type_get<0, type_holder<Ts...> >
00169 {
00170     /// The first type of the type_holder
00171     using type = typename type_holder<Ts...>::head;
00172 };
00173
00174 /**
00175  * @brief      Specialization to recursively pop types to get the kth type.
00176  *
00177  * @tparam     k     Integral constant of the type to get
00178  * @tparam     Ts    List of typenames
00179  */
00180 template <std::size_t k, typename ... Ts>
00181 struct type_get<k, type_holder<Ts...> >
00182 {
00183     /// Recurse after popping the first type off
00184     using type = typename type_get<k-1, typename type_holder<Ts...>::tail>::type;
```

```
00185 };
00186
00187 /// @cond detail
00188 /// Namespace for utility helper functions
00189 namespace detail
00190 {
00191 /**
00192  * @brief      Helper to broadcast a list of types into a class.
00193  *
00194  * @tparam     C     Class to old a list of types
00195  * @tparam     V     Class to broadcast the types into
00196  * @tparam     Rs    List of resulting types
00197  */
00198 template <class C, template <typename> class V, typename ... Rs>
00199 struct type_map_helper {};
00200
00201 /**
00202  * @brief      Terminal condition: place the mapped types into a tuple
00203  *
00204  * @tparam     G     Empty tuple
00205  * @tparam     V     Type template <class T> class to map into
00206  * @tparam     Rs    List of already mapped types
00207  */
00208 template <template <class ...> class G, template <typename> class V, typename ... Rs>
00209 struct type_map_helper<G<>, V, Rs...>
00210 {
00211     using type = G<Rs...>;
00212 };
00213
00214 /**
00215  * @brief      Map types into
00216  *
00217  * @tparam     G     Tuple of types
00218  * @tparam     T     Current type
00219  * @tparam     Ts    List of remaining types
00220  * @tparam     V     Type template <class T> class to map into
00221  * @tparam     Rs    List of already mapped types
00222  */
00223 template <template < class ...> class G, typename T, typename ... Ts, template <typename> class V,
      typename ... Rs>
00224 struct type_map_helper<G<T, Ts...>, V, Rs...>
00225 {
00226     using type = typename type_map_helper<G<Ts...>, V, Rs..., V<T> >::type;
00227 };
00228 }     // end of namespace detail
00229 /// @endcond
00230
00231 /**
00232  * @brief      Map the types in C into `V<T>`.
00233  *
00234  * Given a container of types `C<T1,T2,T3,...>` and template template type
00235  * `V<T>`, this function will apply the types in C to `V<T>`. This produces
00236  * `C<V<T1>, V<T2>, V<T3>, ...>`.
00237  *
00238  * @tparam     C     Container of compile time types.
00239  * @tparam     V     Template template class `V<T>` to map into.
00240  */
00241 template <class C, template <typename> class V>
00242 struct type_map
00243 {
00244     /// Tuple of `C<V<T1>, V<T2>, V<T3>, ...>`
00245     using type = typename detail::type_map_helper<C, V>::type;
00246 };
00247
00248 /// @cond detail
00249 namespace detail
00250 {
00251 /**
00252  * @brief      Template for future specialization
00253  */
00254 template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
      <IntegerType> class F, typename ... Accumulators>
00255 struct int_type_map_helper {};
00256
00257 /**
00258  * @brief      Apply the typenames to the OutHolder
00259  *
00260  * @tparam     Integer      Integral type
00261  * @tparam     OutHolder    Type to ultimately hold the accumulated
00262  * @tparam     InHolder     Class of index sequence
00263  * @tparam     F            Type to apply index to
00264  * @tparam     Accumulator  List of mapped typenames F<I>
00265  */
00266 template <class Integer, template <class ...> class OutHolder, template <class, Integer...> class
      InHolder, template <Integer> class F, class ... Accumulator>
00267 struct int_type_map_helper<Integer, OutHolder, InHolder<Integer>, F, Accumulator...>
00268 {
```

```
00269     using type = OutHolder<Accumulator...>;
00270 };
00271
00272 /**
00273  * @brief      Iterates across integers and fills accumulator with F<I>
00274  *
00275  * @tparam     Integer      Integral type
00276  * @tparam     OutHolder    Type to ultimately hold the accumulated
00277  * @tparam     InHolder     Class of index sequence
00278  * @tparam     I            Current integer
00279  * @tparam     Is           Next integer(s)
00280  * @tparam     F            Type to apply index to
00281  * @tparam     Accumulator  List of previously mapped typenames F<I>
00282  */
00283 template <class Integer, template <class ...> class OutHolder, template <class, Integer...> class
      InHolder, Integer I, Integer... Is, template <Integer> class F, class ... Accumulator>
00284 struct int_type_map_helper<Integer, OutHolder, InHolder<Integer, I, Is...>, F, Accumulator...>
00285 {
00286     using type = typename int_type_map_helper<Integer, OutHolder, InHolder<Integer, Is...>, F,
      Accumulator..., F<I> >::type;
00287 };
00288 } // end namespace detail
00289 /// @endcond
00290
00291 /**
00292  * @brief      Maps an integer sequence and typename, F, into outholder.
00293  *
00294  * Given an Integer Sequence `I<0,1,2,3,...>` and template template type `F<I>`,
00295  * this function produces `Out<F<0>, F<1>, F<2>, ...>`.
00296  *
00297  * @tparam     IntegerType      Typename of an integer type
00298  * @tparam     OutHolder        Typename of a holder for types
00299  * @tparam     IntegerSequence  Integral sequence of types
00300  * @tparam     F                Typename of class to be broadcast with integer
00301  */
00302 template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
      <IntegerType> class F>
00303 struct int_type_map
00304 {
00305     /// Tuple of `Out<F<0>, F<1>, F<2>, ...>`.
00306     using type = typename detail::int_type_map_helper<IntegerType, OutHolder, IntegerSequence,
      F>::type;
00307 };
00308
00309 /**
00310  * @brief      Move a list of types from one container to another.
00311  *
00312  * @tparam     TUPLE        Empty container
00313  * @tparam     HOLDER_FULL  Full container
00314  */
00315 template <template <class ...> class TUPLE, typename HOLDER_FULL>
00316 struct type_swap
00317 {};
00318
00319 /**
00320  * @brief      Move a list of types from one container to another.
00321  *
00322  * @tparam     TUPLE   Empty container
00323  * @tparam     HOLDER  Full container
00324  * @tparam     Ts      Typenames in full container
00325  */
00326 template <template <class ...> class TUPLE, template <class ...> class HOLDER, typename ... Ts>
00327 struct type_swap<TUPLE, HOLDER<Ts...> >
00328 {
00329     /// Empty container filled with typenames from full container
00330     using type = TUPLE<Ts...>;
00331 };
00332
00333 /// @cond detail
00334 namespace detail
00335 {
00336 /**
00337  * @brief      Helper struct to reverse a typed sequence.
00338  *
00339  * @tparam     Integer          Typename of integer class.
00340  * @tparam     IntegerSequence  Sequence of integral types.
00341  * @tparam     Accumulator      Bucket ot hold types.
00342  */
00343 template <class Integer, class IntegerSequence, Integer... Accumulator>
00344 struct reverse_sequence_helper {};
00345
00346 /**
00347  * @brief      Terminal case of typed sequence reversal.
00348  *
00349  * @tparam     Integer      Typename of an integer class
00350  * @tparam     InHolder     Template template type holder
00351  * @tparam     Accumulator  List of reverse ordered typenames.
```

```
00352  */
00353 template <class Integer,
00354           template<class, Integer...> class InHolder,
00355           Integer... Accumulator>
00356 struct reverse_sequence_helper<Integer, InHolder<Integer>, Accumulator...>
00357 {
00358     /// Reversed sequence of types.
00359     using type = InHolder<Integer, Accumulator...>;
00360 };
00361
00362 /**
00363  * @brief      Helper struct to reverse a typed sequence.
00364  *
00365  * @tparam     Integer      Typename of integer class.
00366  * @tparam     InHolder     Type holder.
00367  * @tparam     I            First type in InHolder.
00368  * @tparam     Is           The following types in InHolder.
00369  * @tparam     Accumulator  List of reversed typenames.
00370  */
00371 template <class Integer,
00372           template<class, Integer...> class InHolder,
00373           Integer I, Integer... Is,
00374           Integer... Accumulator>
00375 struct reverse_sequence_helper<Integer, InHolder<Integer, I, Is...>, Accumulator...>
00376 {
00377     // Push the first type into the Accumulator and recurse.
00378     /// Reversed sequence of types.
00379     using type = typename reverse_sequence_helper<Integer,
00380                                                   InHolder<Integer, Is...>, I, Accumulator...>::type;
00381 };
00382 } // end namespace detail
00383 /// @endcond
00384
00385 /**
00386  * @brief      Reverse an Integer Sequence
00387  *
00388  * @tparam     Integer          Typename of an integer class.
00389  * @tparam     IntegerSequence  Sequence of compile-time integers.
00390  */
00391 template <class Integer, class IntegerSequence>
00392 struct reverse_sequence
00393 {
00394     /// Reversed sequence of types.
00395     using type = typename detail::reverse_sequence_helper<Integer, IntegerSequence>::type;
00396 };
00397
00398
00399 /**
00400  * @brief      General template for removing the first value from a type holder.
00401  *
00402  * @tparam     Integer          Typename of integer.
00403  * @tparam     IntegerSequence  Sequence of compile time integers.
00404  */
00405 template <class Integer, class IntegerSequence>
00406 struct remove_first_val {};
00407
00408 /**
00409  * @brief      Specialization for removing first integer from a sequence of
00410  *             compile time integers.
00411  *
00412  * @tparam     Integer   Typename of integer type.
00413  * @tparam     InHolder  Type holder of integer sequence.
00414  * @tparam     I         The first integer
00415  * @tparam     Is        Remaining integers
00416  */
00417 template <class Integer,
00418           template<class, Integer...> class InHolder,
00419           Integer I, Integer... Is>
00420 struct remove_first_val<Integer, InHolder<Integer, I, Is...> >
00421 {
00422     /// Type holder with first value removed.
00423     using type = InHolder<Integer, Is...>;
00424 };
00425
00426 /// @cond detail
00427 namespace detail
00428 {
00429 /**
00430  * @brief      Template type for future specialization
00431
00432  */
00433 template <typename Integer, typename IntegerSequence, typename Fn, typename ... Args>
00434 struct int_for_each_helper {};
00435
00436 /**
00437  * @brief      Terminal Case
00438  *
```

```
00439  * @tparam     Integer   { description }
00440  * @tparam     InHolder  { description }
00441  * @tparam     I         { description }
00442  * @tparam     Fn        { description }
00443  * @tparam     Args      { description }
00444  */
00445 template <class Integer, template <class, Integer...> class InHolder,
00446           Integer I, typename Fn, typename ... Args>
00447 struct int_for_each_helper<Integer, InHolder<Integer, I>, Fn, Args...>
00448 {
00449     static void apply(Fn &&f, Args && ... args)
00450     {
00451         f.template apply<I>(std::forward<Args>(args) ...);
00452     }
00453 };
00454
00455 /**
00456  * @brief      Recurse through the integer series
00457  *
00458  * @tparam     Integer   { description }
00459  * @tparam     InHolder  { description }
00460  * @tparam     I         { description }
00461  * @tparam     Is        { description }
00462  * @tparam     Fn        { description }
00463  * @tparam     Args      { description }
00464  */
00465 template <class Integer, template <class, Integer...> class InHolder,
00466           Integer I, Integer... Is, typename Fn, typename ... Args>
00467 struct int_for_each_helper<Integer, InHolder<Integer, I, Is...>, Fn, Args...>
00468 {
00469     static void apply(Fn &&f, Args && ... args)
00470     {
00471         f.template apply<I>(std::forward<Args>(args) ...);
00472         int_for_each_helper<Integer, InHolder<Integer, Is...>, Fn, Args...>::apply(
00473             std::forward<Fn>(f),
00474             std::forward<Args>(args) ...);
00475     }
00476 };
00477 } // end namespace detail
00478 /// @endcond
00479
00480 /**
00481  * @brief      Calls a function `f.apply<k>()` for a sequence of integer k's
00482  *
00483  * @param[in]  args             Arguments to f
00484  * @param[in]  f                Functor with `apply<k>()` method
00485  *
00486  * @tparam     Integer          Integer type
00487  * @tparam     IntegerSequence  Sequence of integers to iterate
00488  * @tparam     Fn               Typename of functor f
00489  * @tparam     Args             Typenames of the arguments
00490  */
00491 template <class Integer, typename IntegerSequence, typename Fn, typename ... Args>
00492 void int_for_each(Fn &&f, Args && ... args)
00493 {
00494     detail::int_for_each_helper<Integer, IntegerSequence, Fn, Args...>::apply(std::forward<Fn>(f),
00495                                                                               std::forward<Args>(args) ...);
00496 }
00497 } // End of namespace util
```

Index

~simplicial_complex
  casc::simplicial_complex< traits >, 79

AbstractSimplicialComplex
  casc, 26
add_vertex
  casc::simplicial_complex< traits >, 79

begin
  casc::SimplexSet< Complex >, 69
  util::range< T >, 58

casc, 23
  AbstractSimplicialComplex, 26
  check_orientation, 26
  clear_orientation, 27
  compute_orientation, 27
  decimate, 27
  decimateBackHalf, 28
  decimateFirstHalf, 28
  edge_up, 29
  get, 29, 30
  getClosure, 30, 31
  getLink, 31
  getStar, 32
  init_orientation, 33
  kneighbors, 33, 34
  kneighbors_up, 34, 35
  neighbors, 35, 36
  neighbors_up, 36, 37
  operator!=, 37
  operator==, 38
  perform_insertion, 38
  perform_removal, 38
  run_user_callback, 39
  set_difference, 39
  set_intersection, 40
  set_union, 40
  to_string, 41
  visit_BFS_down, 41
  visit_BFS_up, 41
  writeDOT, 42
casc::Orientable, 56
casc::SimplexMap< Complex >, 65
  get, 66, 67
  operator<<, 67
casc::SimplexSet< Complex >, 68
  begin, 69
  cbegin, 70
  cend, 70
  empty, 70
  end, 71
  erase, 71
  find, 72
  insert, 73
  operator<<, 74
  size, 73
casc::simplicial_complex< traits >, 74
  ~simplicial_complex, 79
  add_vertex, 79
  down, 80, 81
  EdgeData, 78
  EdgeID, 98
  eq, 81
  exists, 82
  get_cover, 82, 83
  get_cover_insert, 83
  get_edge_down, 84
  get_edge_up, 85
  get_level, 86
  get_level_id, 86, 87
  get_name, 87, 88
  get_simplex_down, 88, 89
  get_simplex_up, 89–91
  insert, 91, 92
  leq, 93
  lt, 93
  nearBoundary, 94
  NodeData, 79
  onBoundary, 94
  remove, 95
  SimplexID, 98
  size, 96
  up, 96, 97
casc::simplicial_complex< traits >::EdgeID< k >, 50
  down, 52
  EdgeID, 51, 52
  up, 52
casc::simplicial_complex< traits >::SimplexID< k >, 60
  cover, 62
  cover_insert, 63
  get_simplex_up, 63, 64
  indices, 64
  operator<<, 65
  SimplexID, 62
cbegin
  casc::SimplexSet< Complex >, 70
cend
  casc::SimplexSet< Complex >, 70

check_orientation
casc, 26
clear_orientation
casc, 27
compute_orientation
casc, 27
cover
casc::simplicial_complex< traits >::SimplexID< k >, 62
cover_insert
casc::simplicial_complex< traits >::SimplexID< k >, 63

decimate
casc, 27
decimateBackHalf
casc, 28
decimateFirstHalf
casc, 28
down
casc::simplicial_complex< traits >, 80, 81
casc::simplicial_complex< traits >::EdgeID< k >, 52

edge_up
casc, 29
EdgeData
casc::simplicial_complex< traits >, 78
EdgeID
casc::simplicial_complex< traits >, 98
casc::simplicial_complex< traits >::EdgeID< k >, 51, 52
empty
casc::SimplexSet< Complex >, 70
end
casc::SimplexSet< Complex >, 71
util::range< T >, 58
eq
casc::simplicial_complex< traits >, 81
erase
casc::SimplexSet< Complex >, 71
exists
casc::simplicial_complex< traits >, 82

find
casc::SimplexSet< Complex >, 72

get
casc, 29, 30
casc::SimplexMap< Complex >, 66, 67
get_cover
casc::simplicial_complex< traits >, 82, 83
get_cover_insert
casc::simplicial_complex< traits >, 83
get_edge_down
casc::simplicial_complex< traits >, 84
get_edge_up
casc::simplicial_complex< traits >, 85
get_level
casc::simplicial_complex< traits >, 86
get_level_id
casc::simplicial_complex< traits >, 86, 87
get_name
casc::simplicial_complex< traits >, 87, 88
get_simplex_down
casc::simplicial_complex< traits >, 88, 89
get_simplex_up
casc::simplicial_complex< traits >, 89–91
casc::simplicial_complex< traits >::SimplexID< k >, 63, 64
getClosure
casc, 30, 31
getLink
casc, 31
getStar
casc, 32

include/casc/CASCFunctions.h, 105, 106
include/casc/CASCTraversals.h, 111, 113
include/casc/decimate.h, 122, 123
include/casc/index_tracker.h, 130, 132
include/casc/Orientable.h, 143, 144
include/casc/SimplexMap.h, 147, 148
include/casc/SimplexSet.h, 150, 151
include/casc/SimplicialComplex.h, 158, 160
include/casc/stringutil.h, 192
include/casc/typetraits.h, 193, 194
include/casc/util.h, 195, 196
index_tracker, 42
index_tracker::index_tracker< _T, _d >, 54
index_tracker::index_tracker< _T, _d >, 53
index_tracker, 54
index_tracker::index_tracker_detail, 43
index_tracker::index_tracker_detail::BTreeNode< _T, _d >, 49
index_tracker::index_tracker_detail::Interval< T >, 55
operator=, 55
indices
casc::simplicial_complex< traits >::SimplexID< k >, 64
init_orientation
casc, 33
insert
casc::SimplexSet< Complex >, 73
casc::simplicial_complex< traits >, 91, 92
int_for_each
util, 45

kneighbors
casc, 33, 34
kneighbors_up
casc, 34, 35

leq
casc::simplicial_complex< traits >, 93
lt
casc::simplicial_complex< traits >, 93

make_range
util, 46

nearBoundary
casc::simplicial_complex< traits >, 94
neighbors
casc, 35, 36
neighbors_up
casc, 36, 37
NodeData
casc::simplicial_complex< traits >, 79

onBoundary
casc::simplicial_complex< traits >, 94
operator!=
casc, 37
operator<<
casc::SimplexMap< Complex >, 67
casc::SimplexSet< Complex >, 74
casc::simplicial_complex< traits >::SimplexID< k >, 65
operator=
index_tracker::index_tracker_detail::Interval< T >, 55
operator==
casc, 38

perform_insertion
casc, 38
perform_removal
casc, 38

range
util::range< T >, 57
remove
casc::simplicial_complex< traits >, 95
run_user_callback
casc, 39

set_difference
casc, 39
set_intersection
casc, 40
set_union
casc, 40
SimplexID
casc::simplicial_complex< traits >, 98
casc::simplicial_complex< traits >::SimplexID< k >, 62
size
casc::SimplexSet< Complex >, 73
casc::simplicial_complex< traits >, 96

to_string
casc, 41
type_name
typetraits.h, 194
typetraits.h
type_name, 194

up
casc::simplicial_complex< traits >, 96, 97
casc::simplicial_complex< traits >::EdgeID< k >, 52
util, 44
int_for_each, 45
make_range, 46
util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >, 54
util::range< T >, 56
begin, 58
end, 58
range, 57
util::remove_first_val< Integer, InHolder< Integer, I, Is... > >, 59
util::remove_first_val< Integer, IntegerSequence >, 58
util::reverse_sequence< Integer, IntegerSequence >, 59
util::type_get< 0, type_holder< Ts... > >, 99
util::type_get< k, T >, 98
util::type_get< k, type_holder< Ts... > >, 99
util::type_holder< T, Ts... >, 100
util::type_holder< Ts >, 100
util::type_map< C, V >, 101
util::type_swap< TUPLE, HOLDER< Ts... > >, 102
util::type_swap< TUPLE, HOLDER_FULL >, 102

visit_BFS_down
casc, 41
visit_BFS_up
casc, 41

writeDOT
casc, 42