

CASC

1.0.5

Generated by Doxygen 1.11.0

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
8.1.1.2 NodeSet	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]	30
8.1.2.13 getLink() [1/2]	31
8.1.2.14 getLink() [2/2]	31
8.1.2.15 getStar() [1/2]	32
8.1.2.16 getStar() [2/2]	32
8.1.2.17 init_orientation()	32
8.1.2.18 kneighbors() [1/2]	33
8.1.2.19 kneighbors() [2/2]	33
8.1.2.20 kneighbors_up() [1/2]	34
8.1.2.21 kneighbors_up() [2/2]	34
8.1.2.22 neighbors() [1/2]	34
8.1.2.23 neighbors() [2/2]	35
8.1.2.24 neighbors_up() [1/2]	35
8.1.2.25 neighbors_up() [2/2]	36
8.1.2.26 operator!=(())	36
8.1.2.27 operator==(())	36
8.1.2.28 perform_insertion()	37
8.1.2.29 perform_removal()	37
8.1.2.30 run_user_callback()	38
8.1.2.31 set_difference()	39
8.1.2.32 set_intersection()	39
8.1.2.33 set_union()	40
8.1.2.34 to_string()	40
8.1.2.35 visit_BFS_down()	40
8.1.2.36 visit_BFS_up()	41
8.1.2.37 writeDOT()	41
8.2 index_tracker Namespace Reference	41
8.3 index_tracker::index_tracker_detail Namespace Reference	42
8.4 util Namespace Reference	44
8.4.1 Function Documentation	44
8.4.1.1 int_for_each()	44
8.4.1.2 make_range() [1/2]	45
8.4.1.3 make_range() [2/2]	45
9 Data Structure Documentation	47
9.1 index_tracker::index_tracker_detail::BTreeNode< _T, _d > Struct Template Reference	47
9.1.1 Detailed Description	47
9.2 casc::simplicial_complex< traits >::EdgeID< k > Struct Template Reference	48
9.2.1 Detailed Description	49

9.2.2 Constructor & Destructor Documentation	49
9.2.2.1 EdgeID() [1/2]	49
9.2.2.2 EdgeID() [2/2]	50
9.2.3 Member Function Documentation	50
9.2.3.1 down()	50
9.2.3.2 up()	50
9.3 index_tracker::index_tracker< _T, _d > Class Template Reference	50
9.3.1 Detailed Description	51
9.3.2 Constructor & Destructor Documentation	51
9.3.2.1 index_tracker()	51
9.4 util::int_type_map< IntegerType, OutHolder, IntegerSequence, F > Struct Template Reference	52
9.4.1 Detailed Description	52
9.5 index_tracker::index_tracker_detail::Interval< T > Struct Template Reference	52
9.5.1 Detailed Description	53
9.5.2 Member Function Documentation	53
9.5.2.1 operator=()	53
9.6 casc::Orientable Struct Reference	54
9.7 util::range< T > Struct Template Reference	54
9.7.1 Detailed Description	54
9.7.2 Constructor & Destructor Documentation	54
9.7.2.1 range() [1/2]	54
9.7.2.2 range() [2/2]	55
9.7.3 Member Function Documentation	55
9.7.3.1 begin()	55
9.7.3.2 end()	55
9.8 util::remove_first_val< Integer, IntegerSequence > Struct Template Reference	56
9.8.1 Detailed Description	56
9.9 util::remove_first_val< Integer, InHolder< Integer, I, Is... > > Struct Template Reference	56
9.9.1 Detailed Description	56
9.10 util::reverse_sequence< Integer, IntegerSequence > Struct Template Reference	57
9.10.1 Detailed Description	57
9.11 casc::simplicial_complex< traits >::SimplexID< k > Struct Template Reference	57
9.11.1 Detailed Description	59
9.11.2 Constructor & Destructor Documentation	59
9.11.2.1 SimplexID() [1/2]	59
9.11.2.2 SimplexID() [2/2]	59
9.11.3 Member Function Documentation	60
9.11.3.1 cover()	60
9.11.3.2 cover_insert()	60
9.11.3.3 get_simplex_up() [1/3]	60
9.11.3.4 get_simplex_up() [2/3]	61
9.11.3.5 get_simplex_up() [3/3]	61

9.11.3.6 indices()	61
9.11.4 Friends And Related Symbol Documentation	62
9.11.4.1 operator<<	62
9.12 casc::SimplexMap< Complex > Struct Template Reference	62
9.12.1 Detailed Description	63
9.12.2 Member Typedef Documentation	63
9.12.2.1 cLevelIndex	63
9.12.2.2 cRevIndex	64
9.12.2.3 RevIndex	64
9.12.3 Member Function Documentation	64
9.12.3.1 get() [1/2]	64
9.12.3.2 get() [2/2]	64
9.12.4 Friends And Related Symbol Documentation	64
9.12.4.1 operator<<	64
9.13 casc::SimplexSet< Complex > Struct Template Reference	65
9.13.1 Detailed Description	67
9.13.2 Member Typedef Documentation	67
9.13.2.1 cLevelIndex	67
9.13.2.2 cRevIndex	67
9.13.2.3 RevIndex	67
9.13.2.4 SimplexIDLevel	68
9.13.3 Member Function Documentation	68
9.13.3.1 begin()	68
9.13.3.2 cbegin()	68
9.13.3.3 cend()	68
9.13.3.4 empty()	69
9.13.3.5 end()	69
9.13.3.6 erase() [1/2]	69
9.13.3.7 erase() [2/2]	69
9.13.3.8 find() [1/2]	70
9.13.3.9 find() [2/2]	70
9.13.3.10 get() [1/2]	70
9.13.3.11 get() [2/2]	71
9.13.3.12 insert() [1/2]	71
9.13.3.13 insert() [2/2]	71
9.13.3.14 size()	71
9.13.4 Friends And Related Symbol Documentation	72
9.13.4.1 operator<<	72
9.14 casc::simplicial_complex< traits > Class Template Reference	72
9.14.1 Detailed Description	76
9.14.2 Member Typedef Documentation	76
9.14.2.1 EdgeData	76

9.14.2.2 NodeData	77
9.14.3 Constructor & Destructor Documentation	77
9.14.3.1 ~simplicial_complex()	77
9.14.4 Member Function Documentation	77
9.14.4.1 add_vertex() [1/2]	77
9.14.4.2 add_vertex() [2/2]	77
9.14.4.3 down() [1/3]	77
9.14.4.4 down() [2/3]	78
9.14.4.5 down() [3/3]	78
9.14.4.6 eq() [1/2]	79
9.14.4.7 eq() [2/2]	79
9.14.4.8 exists()	79
9.14.4.9 get_cover() [1/2]	80
9.14.4.10 get_cover() [2/2]	80
9.14.4.11 get_cover_insert()	81
9.14.4.12 get_edge_down() [1/2]	81
9.14.4.13 get_edge_down() [2/2]	81
9.14.4.14 get_edge_up() [1/2]	82
9.14.4.15 get_edge_up() [2/2]	82
9.14.4.16 get_level() [1/2]	83
9.14.4.17 get_level() [2/2]	83
9.14.4.18 get_level_id() [1/2]	83
9.14.4.19 get_level_id() [2/2]	83
9.14.4.20 get_name() [1/3]	84
9.14.4.21 get_name() [2/3]	84
9.14.4.22 get_name() [3/3]	84
9.14.4.23 get_simplex_down() [1/3]	85
9.14.4.24 get_simplex_down() [2/3]	85
9.14.4.25 get_simplex_down() [3/3]	85
9.14.4.26 get_simplex_up() [1/4]	86
9.14.4.27 get_simplex_up() [2/4]	86
9.14.4.28 get_simplex_up() [3/4]	86
9.14.4.29 get_simplex_up() [4/4]	87
9.14.4.30 insert() [1/4]	87
9.14.4.31 insert() [2/4]	88
9.14.4.32 insert() [3/4]	88
9.14.4.33 insert() [4/4]	88
9.14.4.34 leq()	89
9.14.4.35 lt()	90
9.14.4.36 nearBoundary()	91
9.14.4.37 onBoundary()	91
9.14.4.38 remove() [1/3]	91

9.14.4.39 remove() [2/3]	92
9.14.4.40 remove() [3/3]	92
9.14.4.41 size()	93
9.14.4.42 up() [1/3]	94
9.14.4.43 up() [2/3]	94
9.14.4.44 up() [3/3]	94
9.14.5 Friends And Related Symbol Documentation	95
9.14.5.1 EdgeID	95
9.14.5.2 SimplexID	95
9.15 util::type_get< k, T > Struct Template Reference	95
9.15.1 Detailed Description	95
9.16 util::type_get< 0, type_holder< Ts... > > Struct Template Reference	96
9.16.1 Detailed Description	96
9.17 util::type_get< k, type_holder< Ts... > > Struct Template Reference	96
9.17.1 Detailed Description	96
9.18 util::type_holder< Ts > Struct Template Reference	97
9.18.1 Detailed Description	97
9.19 util::type_holder< T, Ts... > Struct Template Reference	97
9.19.1 Detailed Description	98
9.20 util::type_map< C, V > Struct Template Reference	98
9.20.1 Detailed Description	98
9.21 util::type_swap< TUPLE, HOLDER_FULL > Struct Template Reference	99
9.21.1 Detailed Description	99
9.22 util::type_swap< TUPLE, HOLDER< Ts... > > Struct Template Reference	99
9.22.1 Detailed Description	99
10 File Documentation	101
10.1 include/casc/CASCFunctions.h File Reference	101
10.2 CASCFunctions.h	102
10.3 include/casc/CASCTraversals.h File Reference	105
10.4 CASCTraversals.h	106
10.5 include/casc/decimate.h File Reference	112
10.6 decimate.h	113
10.7 include/casc/index_tracker.h File Reference	118
10.8 index_tracker.h	120
10.9 include/casc/Orientable.h File Reference	131
10.10 Orientable.h	132
10.11 include/casc/SimplexMap.h File Reference	134
10.12 SimplexMap.h	135
10.13 include/casc/SimplexSet.h File Reference	136
10.14 SimplexSet.h	137
10.15 include/casc/SimplicialComplex.h File Reference	142

10.16 SimplicialComplex.h	143
10.17 include/casc/stringutil.h File Reference	161
10.18 stringutil.h	162
10.19 include/casc/typetraits.h File Reference	162
10.19.1 Detailed Description	162
10.19.2 Function Documentation	163
10.19.2.1 type_name()	163
10.20 typetraits.h	163
10.21 include/casc/util.h File Reference	164
10.22 util.h	165
Index	169

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 wish 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

This project is licensed under the GNU Lesser General Public License v2.1 - please see the [COPYING.md](#) file for details.

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	41
index_tracker::index_tracker_detail	B-tree internal data structures	42
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	47
casc::simplicial_complex< traits >::EdgeID< k >	
External reference to an edge or a connection within the complex	48
index_tracker::index_tracker< _T, _d >	
Tracker of available indices implemented as a B-tree of intervals	50
util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >	
Maps an integer sequence and typename, F, into outholder	52
index_tracker::index_tracker_detail::Interval< T >	
Interval object represents a range	52
casc::Orientable	
Class representing the orientation	54
util::range< T >	
A range object to support range based for loops	54
util::remove_first_val< Integer, IntegerSequence >	
General template for removing the first value from a type holder	56
util::remove_first_val< Integer, InHolder< Integer, I, Is... > >	
Specialization for removing first integer from a sequence of compile time integers	56
util::reverse_sequence< Integer, IntegerSequence >	
Reverse an Integer Sequence	57
casc::simplicial_complex< traits >::SimplexID< k >	
A handle for a simplex object in the complex	57
casc::SimplexMap< Complex >	
A multimap to represent a map of simplex indices to a set of simplices	62
casc::SimplexSet< Complex >	
A multiset to store simplices in a simplicial_complex	65
casc::simplicial_complex< traits >	
The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring	72
util::type_get< k, T >	
Helper to get the kth element from a type_holder	95
util::type_get< 0, type_holder< Ts... > >	
Specialization for terminal case	96
util::type_get< k, type_holder< Ts... > >	
Specialization to recursively pop types to get the kth type	96

util::type_holder< Ts >	
Queue based data structure to hold list of types	97
util::type_holder< T, Ts... >	
Partial specialization to allow FIFO access of typenames	97
util::type_map< C, V >	
Map the types in C into V<T>	98
util::type_swap< TUPLE, HOLDER_FULL >	
Move a list of types from one container to another	99
util::type_swap< TUPLE, HOLDER< Ts... > >	
Move a list of types from one container to another	99

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	101
include/casc/CASCTraversals.h	
Implementations of various advanced traversals such as by neighborhood and breadth first search	105
include/casc/decimate.h	
Meta-data aware decimation functions	112
include/casc/index_tracker.h	
B-tree based interval tracker	118
include/casc/Orientable.h	
Data type for orientability	131
include/casc/SimplexMap.h	
SimplexMap data structure and associated convenience functions	134
include/casc/SimplexSet.h	
SimplexSet data structure and associated convenience functions	136
include/casc/SimplicialComplex.h	
This header contains the main CASC data structure and associated components	142
include/casc/stringutil.h	
String utilities for CASC	161
include/casc/typetraits.h	
Helper functions for debugging template types	162
include/casc/util.h	
Metatemplate pack expansion helpers	164

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](#)
- template<typename T >
using [NodeSet](#)
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
```

Initial value:

```
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
>();
```

8.1.1.2 NodeSet

```
template<typename T >
using casc::NodeSet
```

Initial value:

```
std::unordered_set<T, simplex_set_detail::hashSimplexID<T> >
```

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

<i>F</i>	Simplicial_complex
----------	--------------------

Template Parameters

<i>Complex</i>	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

<i>F</i>	Simplicial complex of interest
----------	--------------------------------

Template Parameters

<i>Complex</i>	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

<i>F</i>	Simplicial_complex
----------	--------------------

Template Parameters

<i>Complex</i>	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	<i>F</i>	simplicial_complex to operate on.
in	<i>s</i>	Simplex to decimate.
in	<i>clbk</i>	Callback function to map meta-data

Template Parameters

<i>Complex</i>	Typename of the simplicial_complex
<i>Simplex</i>	Typename of the simplex
<i>Callback</i>	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

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

Template Parameters

<i>Complex</i>	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	<i>F</i>	simplicial_complex to operate on.
in	<i>s</i>	Simplex to decimate.
	<i>simplexMap</i>	The simplex map to populate

Template Parameters

<i>Complex</i>	Typename of the simplicial_complex
<i>Simplex</i>	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	<i>v</i>	Visitor functor to apply.
	<i>F</i>	The simplicial_complex to traverse.
in	<i>s</i>	The edge to start at.

Template Parameters

<i>Visitor</i>	Typename of the functor.
<i>EdgeID</i>	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

<i>S</i>	SimplexMap to retrieve from.
----------	----------------------------------------------

Template Parameters

k	Simplex dimension.
<i>Complex</i>	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

<i>S</i>	SimplexSet of interest.
----------	-----------------------------------------

Template Parameters

k	Simplex dimension desired.
<i>Complex</i>	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	<i>F</i>	Complex of interest.
in	<i>S</i>	SimplexSet to compute the closure of.
out	<i>dest</i>	Destination SimplexSet

Template Parameters

<i>Complex</i>	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	<i>F</i>	Complex of interest.
in	<i>s</i>	Simplex of interest.
out	<i>dest</i>	Destination SimplexSet .

Template Parameters

<i>Complex</i>	Typename of the complex.
<i>Simplex</i>	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	<i>F</i>	Complex of interest.
in	<i>S</i>	SimplexSet to get the link of.
out	<i>dest</i>	Destination SimplexSet .

Template Parameters

<i>Complex</i>	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

<i>F</i>	Complex of interest.
<i>s</i>	Simplex of interest.
<i>dest</i>	Destination SimplexSet .

Template Parameters

<i>Complex</i>	Typename of the complex.
<i>Simplex</i>	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	<i>F</i>	Complex of interest.
in	<i>S</i>	SimplexSet to compute the star of.
out	<i>dest</i>	Destination SimplexSet.

Template Parameters

<i>Complex</i>	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	<i>F</i>	Complex of interest.
	<i>s</i>	Simplex to get the star of.
out	<i>dest</i>	Destination SimplexSet.

Template Parameters

<i>Complex</i>	Typename of the complex.
<i>Simplex</i>	Typename of the simplex.

8.1.2.17 init_orientation()

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

Parameters

<i>F</i>	Simplicial complex of interest
----------	--------------------------------

Template Parameters

<i>Complex</i>	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	<i>F</i>	The simplicial_complex to traverse.
in	<i>ring</i>	The number of rings of neighbors to collect.
out	<i>nbors</i>	Set of previously seen simplices.
in	<i>begin</i>	The begin
in	<i>end</i>	The end

Template Parameters

<i>Complex</i>	Typename of the simplicial_complex .
<i>level</i>	Simplex dimension of the simplex and neighbors.
<i>Iterator</i>	{ 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	<i>F</i>	The simplicial complex
in	<i>nid</i>	Simplex of interest to get the neighbors of.
in	<i>ring</i>	The number of rings to include as a neighbor.
out	<i>nbors</i>	Set of neighbors to populate.

Template Parameters

<i>Complex</i>	Typename of the complex.
<i>SimplexID</i>	Typename of the SimplexID.

8.1.2.20 neighbors_up() [1/2]

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

Parameters

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

Template Parameters

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

8.1.2.21 neighbors_up() [2/2]

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

Parameters

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

Template Parameters

<i>Complex</i>	Typename of the complex.
<i>SimplexID</i>	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

	<i>F</i>	The simplicial complex.
in	<i>nid</i>	Simplex to get neighbors of.
in	<i>iter</i>	The iterator to push members into.

Template Parameters

<i>Complex</i>	Type of the simplicial complex
<i>level</i>	The integral level of the node
<i>InsertIter</i>	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

	<i>F</i>	The simplicial complex
in	<i>nid</i>	Simplex to get neighbors of.
in	<i>iter</i>	The iterator to push members into.

Template Parameters

<i>Complex</i>	Type of the simplicial complex
<i>level</i>	The integral level of the node
<i>InsertIter</i>	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

	<i>F</i>	The simplicial complex.
in	<i>nid</i>	Simplex to get neighbors of.
in	<i>iter</i>	The iterator to push members into.

Template Parameters

<i>Complex</i>	Type of the simplicial complex
<i>level</i>	The integral level of the node
<i>InsertIter</i>	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

	<i>F</i>	The simplicial complex.
in	<i>nid</i>	Simplex to get neighbors of.
in	<i>iter</i>	The iterator to push members into.

Template Parameters

<i>Complex</i>	Type of the simplicial complex
<i>level</i>	The integral level of the node
<i>InsertIter</i>	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	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>Complex</i>	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	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>Complex</i>	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

<i>F</i>	The simplicial_complex to insert into.
<i>S</i>	SimplexSet of simplices to insert.

Template Parameters

<i>Complex</i>	Typename of complex
----------------	---------------------

8.1.2.29 perform_removal()

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

Parameters

<i>F</i>	The simplicial_complex to remove from.
<i>S</i>	SimplexSet of simplices to remove.

Template Parameters

<i>Complex</i>	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	<i>F</i>	The simplicial_complex
in	<i>S</i>	SimplexMap of
in	<i>clbk</i>	User specified callback functor
out	<i>rv</i>	Multi-vector to place results.

Template Parameters

<i>Complex</i>	Typename of the simplicial_complex
<i>Callback</i>	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	<i>A</i>	A SimplexSet
in	<i>B</i>	Another SimplexSet
out	<i>dest</i>	The destination SimplexSet .

Template Parameters

<i>Complex</i>	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	<i>A</i>	A SimplexSet
in	<i>B</i>	Another SimplexSet
out	<i>dest</i>	The destination SimplexSet .

Template Parameters

<i>Complex</i>	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	<i>A</i>	A SimplexSet
in	<i>B</i>	Another SimplexSet
out	<i>dest</i>	The destination SimplexSet .

Template Parameters

<i>Complex</i>	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	<i>A</i>	Array containing name of a simplex.
----	----------	-------------------------------------

Template Parameters

<i>T</i>	Typename KeyType.
<i>k</i>	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	<i>v</i>	Visitor functor to apply.
	<i>F</i>	The simplicial_complex to traverse.
in	<i>s</i>	The simplex to start at.

Template Parameters

<i>Visitor</i>	Typename of the functor.
<i>SimplexID</i>	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	<i>v</i>	Visitor functor to apply.
	<i>F</i>	The simplicial_complex to traverse.
in	<i>s</i>	The simplex to start at.

Template Parameters

<i>Visitor</i>	Typename of the functor.
<i>SimplexID</i>	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	<i>filename</i>	Filename to write out to.
in	<i>F</i>	Simplicial complex to generate the DOT of.

Template Parameters

<i>Complex</i>	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	<i>args</i>	Arguments to f
in	<i>f</i>	Functor with <code>apply<k>()</code> method

Template Parameters

<i>Integer</i>	Integer type
<i>IntegerSequence</i>	Sequence of integers to iterate
<i>Fn</i>	Typename of functor f
<i>Args</i>	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	<i>p</i>	A pair containing begin and end iterators.
----	----------	--------------------------------------------

Template Parameters

<i>T</i>	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	<i>b</i>	Iterator to the beginning.
in	<i>e</i>	Iterator to the end.

Template Parameters

<i>T</i>	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

\leftrightarrow	{ description }
$\overleftarrow{\leftrightarrow}$	
T	
\leftrightarrow	{ description }
$\overleftarrow{\leftrightarrow}$	
d	

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 constructor 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

<code>k</code>	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	<code>p</code>	Pointer to the next Node.
in	<code>e</code>	Key of the edge

9.2.2.2 EdgelD() [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

\leftarrow $_T$	Typename of the indices
\leftarrow $_d$	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 $I<0, 1, 2, 3, \dots>$ and template template type $F<I>$, this function produces $Out<F<0>, F<1>, F<2>, \dots>$.

Template Parameters

<i>IntegerType</i>	Typename of an integer type
<i>OutHolder</i>	Typename of a holder for types
<i>IntegerSequence</i>	Integral sequence of types
<i>F</i>	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)
Assignment 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

<i>T</i>	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

<i>in</i>	<i>rhs</i>	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 looping added in C++11. See also `range-for`.

Template Parameters

<code>T</code>	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	<i>c</i>	Container class which implements begin() and end() .
----	----------	--------------------------------------------------------------------------------------

Template Parameters

<i>C</i>	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	<i>b</i>	Beginning iterator
in	<i>e</i>	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

<i>Integer</i>	Typename of integer.
<i>IntegerSequence</i>	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

<i>Integer</i>	Typename of integer type.
<i>InHolder</i>	Type holder of integer sequence.
<i>I</i>	The first integer
<i>Is</i>	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

<i>Integer</i>	Typename of an integer class.
<i>IntegerSequence</i>	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

<code>k</code>	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

<code>in</code>	<code>p</code>	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	<i>rhs</i>	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	<i>pos</i>	Iterator inserter
----	------------	-------------------

Template Parameters

<i>Inserter</i>	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	<i>id</i>	The identifier of a simplex.
in	<i>s</i>	The relative single character name of the desired simplex.

Template Parameters

<i>i</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	<i>s</i>	Array of keys to follow
----	----------	-------------------------

Template Parameters

<i>j</i>	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	<i>arr</i>	Array of keys to follow
----	------------	-------------------------

Template Parameters

<i>j</i>	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	<i>id</i>	SimplexID of the simplex of interest.
----	-----------	-------------------------------------------------------

Returns

Array containing the name of 'id'.

9.11.4 Friends And Related Symbol 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

	<i>out</i>	Handle to the stream
in	<i>nid</i>	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**
Index sequence starting at 1.
- using **RevIndex**
Reversed Index sequence.
- using **cRevIndex**
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

<i>Complex</i>	Typename of the <code>simplicial_complex</code> .
----------------	---------------------------------------------------

9.12.2 Member Typedef Documentation

9.12.2.1 cLevelIndex

```
template<typename Complex >
using casc::SimplexMap< Complex >::cLevelIndex
```

Initial value:

```
typename util::remove_first_val<std::size_t,
                                LevelIndex>::type
```

9.12.2.2 cRevIndex

```
template<typename Complex >
using casc::SimplexMap< Complex >::cRevIndex
```

Initial value:

```
typename util::reverse_sequence<std::size_t,
                                cLevelIndex>::type
```

9.12.2.3 RevIndex

```
template<typename Complex >
using casc::SimplexMap< Complex >::RevIndex
```

Initial value:

```
typename util::reverse_sequence<std::size_t,
                                LevelIndex>::type
```

9.12.3 Member Function Documentation

9.12.3.1 get() [1/2]

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

Template Parameters

<i>k</i>	Simplex dimension to retrieve.
----------	--------------------------------

Returns

A map of SimplexID<k> to [SimplexSet](#).

9.12.3.2 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.

9.12.4 Friends And Related Symbol Documentation

9.12.4.1 operator<<

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

Parameters

	<i>output</i>	Handle to the stream to print to.
<i>in</i>	<i>S</i>	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](#)
Index sequence starting at 1.
- using [RevIndex](#)
Reversed index sequence.
- using [cRevIndex](#)
Reversed index sequence stops at 1.
- using **type_this** = `SimplexSet<Complex>`
Typename of this.
- using [SimplexIDLevel](#)
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 iterator to the first element of the container.
- template<std::size_t k>
auto **cbegin** () const
Get an iterator to the first element of the container.
- template<std::size_t k>
auto & **get** ()
Get the NodeSet for a particular simplex dimension.
- template<std::size_t k>
auto & **get** () const
Get the NodeSet for a particular simplex dimension.

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

<i>Complex</i>	Typename of the simplicial_complex .
----------------	------------------------------------------------------

9.13.2 Member Typedef Documentation

9.13.2.1 `cLevelIndex`

```
template<typename Complex >
using casc::SimplexSet< Complex >::cLevelIndex
```

Initial value:

```
typename util::remove_first_val<std::size_t,
                                LevelIndex>::type
```

9.13.2.2 `cRevIndex`

```
template<typename Complex >
using casc::SimplexSet< Complex >::cRevIndex
```

Initial value:

```
typename util::reverse_sequence<std::size_t,
                                cLevelIndex>::type
```

9.13.2.3 `RevIndex`

```
template<typename Complex >
using casc::SimplexSet< Complex >::RevIndex
```

Initial value:

```
typename util::reverse_sequence<std::size_t,
                                LevelIndex>::type
```

9.13.2.4 SimplexIDLevel

```
template<typename Complex >
using casc::SimplexSet< Complex >::SimplexIDLevel
```

Initial value:

```
typename util::int_type_map<std::size_t,
                                std::tuple, LevelIndex, SimplexID>::type
```

9.13.3 Member Function Documentation

9.13.3.1 begin()

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

Template Parameters

<i>k</i>	The simplex dimension to get iterator of.
----------	-------------------------------------------

Returns

Returns an iterator to the first element.

9.13.3.2 cbegin()

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

Template Parameters

<i>k</i>	The simplex dimension to get iterator of.
----------	-------------------------------------------

Returns

Returns an iterator to the first element.

9.13.3.3 cend()

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

Template Parameters

<i>k</i>	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.

9.13.3.4 `empty()`

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

Template Parameters

<code>k</code>	Level to check.
----------------	-----------------

Returns

True if the container is empty, false otherwise.

9.13.3.5 `end()`

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

Template Parameters

<code>k</code>	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.

9.13.3.6 `erase()` [1/2]

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

Parameters

<code>in</code>	<code>s</code>	<code>SimplexSet</code> to remove.
-----------------	----------------	------------------------------------

9.13.3.7 `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'.
---	---------------------------

9.13.3.8 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.

9.13.3.9 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.

9.13.3.10 get() [1/2]

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


Template Parameters

<i>k</i>	Simplex dimension to get.
----------	---------------------------

Returns

Returns the NodeSet corresponding to the requested dimension.

9.13.3.11 `get()` [2/2]

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

Template Parameters

<i>k</i>	Simplex dimension to get.
----------	---------------------------

Returns

Returns the NodeSet corresponding to the requested dimension.

9.13.3.12 `insert()` [1/2]

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

Parameters

in	<i>s</i>	The <code>SimplexSet</code> to insert.
----	----------	----------------------------------------

9.13.3.13 `insert()` [2/2]

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

Parameters

in	<i>s</i>	Simplex to insert.
----	----------	--------------------

Template Parameters

<i>k</i>	Simplex dimension of 's'.
----------	---------------------------

9.13.3.14 `size()`

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

Template Parameters

<i>k</i>	Simplex dimension to query
----------	----------------------------

Returns

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

9.13.4 Friends And Related Symbol Documentation

9.13.4.1 operator<<

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

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

Parameters

	<i>output</i>	Handle to the stream to print to.
<i>in</i>	<i>S</i>	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 Insertter >`
`void get_cover_insert (const SimplexID< k > id, Insertter pos) const`
Insert the coboundary keys of a simple into an insertter.
- `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`

Checks whether a simplex is on a boundary.

- `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

<i>traits</i>	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

<i>simplex</i>	The simplex of interest.
----------------	--------------------------

Template Parameters

<i>k</i>	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

<i>simplices</i>	The set of simplicies.
------------------	------------------------

Template Parameters

<i>k</i>	The dimension of the simplicies.
----------	----------------------------------

Returns

The set of boundary simplicies.

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

<i>simplices</i>	The set of simplicies.
------------------	------------------------

Template Parameters

<i>k</i>	The dimension of the simplicies.
----------	----------------------------------

Returns

The set of boundary simplicies.

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	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>k</i>	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	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>L</i>	Dimension of lhs simplex.
<i>R</i>	Dimension of rhs simplex.

Returns

Always false as $L \neq R$. 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	<i>s</i>	C-style array of the name
----	----------	---------------------------

Template Parameters

<i>k</i>	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	<i>id</i>	The identifier of a simplex.
----	-----------	------------------------------

Template Parameters

<i>k</i>	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	<i>id</i>	The identifier
in	<i>fn</i>	The function

Template Parameters

<i>k</i>	The dimension of the simplex.
<i>Lambda</i>	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	<i>id</i>	The identifier of a simplex.
in	<i>pos</i>	Iterator inserter

Template Parameters

<i>k</i>	The dimension of the simplex.
<i>Inserter</i>	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	<i>simplex</i>	The simplex of interest.
in	<i>a</i>	Key of the edge to get.

Template Parameters

<i>k</i>	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	<i>simplex</i>	The simplex of interest.
in	<i>a</i>	Key of the edge to get.

Template Parameters

<i>k</i>	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	<i>simplex</i>	The simplex of interest.
in	<i>a</i>	Key of the edge to get.

Template Parameters

<i>k</i>	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	<i>simplex</i>	The simplex of interest.
in	<i>a</i>	Key of the edge to get.

Template Parameters

<code>k</code>	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

<code>k</code>	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

<code>k</code>	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

<code>k</code>	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

<i>k</i>	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	<i>id</i>	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	<i>id</i>	SimplexID of the simplex of interest.
----	-----------	-------------------------------------------------------

Template Parameters

<i>n</i>	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	<i>id</i>	SimplexID of the simplex of interest.
in	<i>fn</i>	Lambda function to apply to the name of 'id'.

Template Parameters

<i>n</i>	Dimension of simplex 'id'.
<i>Lambda</i>	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	<i>id</i>	The identifier of a simplex.
in	<i>s</i>	The relative single character name of the desired simplex.

Template Parameters

<i>i</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	<i>id</i>	The identifier of a simplex.
in	<i>s</i>	The relative name of the desired simplex.

Template Parameters

<i>i</i>	The size of simplex 'id'.
<i>j</i>	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	<i>s</i>	Name of the simplex to find.
----	----------	------------------------------

Template Parameters

<i>n</i>	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	<i>id</i>	The identifier of a simplex.
in	<i>s</i>	The relative single character name of the desired simplex.

Template Parameters

<i>i</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	<i>id</i>	The identifier of a simplex.
in	<i>s</i>	The relative name of the desired simplex.

Template Parameters

<i>i</i>	The size of simplex 'id'.
<i>j</i>	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	<i>s</i>	A C style array of vertices of simplex 's'.
----	----------	---------------------------------------------

Template Parameters

<i>n</i>	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	<i>s</i>	A C style array of vertices of simplex 's'.
in	<i>data</i>	The data to be stored at the simplex 's'.

Template Parameters

<i>n</i>	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	<i>s</i>	Array of vertices comprising 's'.
----	----------	-----------------------------------

Template Parameters

<i>n</i>	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

<code>in</code>	<code>s</code>	Array of vertices comprising 's'.
<code>in</code>	<code>data</code>	The data to be stored at the simplex 's'.

Template Parameters

<code>n</code>	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

<code>in</code>	<code>s</code>	SimplexID of interest
-----------------	----------------	---------------------------------------

Template Parameters

<code>k</code>	Dimension of the simplex
----------------	--------------------------

Returns

True if the simplex interacts with a topLevel-1 simplex which is on a boundary.

Specialization of the facets

Parameters

<code>in</code>	<code>s</code>	SimplexID of interest
-----------------	----------------	---------------------------------------

Template Parameters

<code>k</code>	Dimension of the simplex
----------------	--------------------------

Returns

True if s is on a boundary

Specialization of the topLevel-1 simplices

Parameters

in	<i>s</i>	SimplexID of interest
----	----------	-----------------------

Template Parameters

<i>k</i>	Dimension of the simplex
----------	--------------------------

Returns

True if *s* is on a boundary

Less than or equal to comparison operator of two SimplexIDs.

Parameters

in	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>L</i>	Dimension of lhs simplex.
<i>R</i>	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	<i>lhs</i>	The left hand side
in	<i>rhs</i>	The right hand side

Template Parameters

<i>L</i>	Dimension of lhs simplex.
<i>R</i>	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

<code>in</code>	<code>s</code>	<code>SimplexID</code> of interest
-----------------	----------------	------------------------------------

Template Parameters

<code>level</code>	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

<code>in</code>	<code>s</code>	<code>SimplexID</code> of interest
-----------------	----------------	------------------------------------

Template Parameters

<code>k</code>	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

<i>k</i>	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

<i>nid</i>	The simplex of interest
------------	-------------------------

Template Parameters

<i>k</i>	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

<i>simplices</i>	The set of simplices
------------------	----------------------

Template Parameters

<i>k</i>	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

<i>simplices</i>	The set of simplices
------------------	----------------------

Template Parameters

<i>k</i>	The dimension of the simplices.
----------	---------------------------------

Returns

The set of coboundary simplices.

9.14.5 Friends And Related Symbol 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

<i>k</i>	Integer index of the type to retrieve
<i>T</i>	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

<i>Ts</i>	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

<i>k</i>	Integral constant of the type to get
<i>Ts</i>	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

<i>Ts</i>	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

<i>T</i>	The first typename
<i>Ts</i>	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 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

<i>C</i>	Container of compile time types.
<i>V</i>	Template template class $\mathbb{V}<\mathbb{T}>$ 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

<i>TUPLE</i>	Empty container
<i>HOLDER_FULL</i>	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

<i>TUPLE</i>	Empty container
<i>HOLDER</i>	Full container
<i>Ts</i>	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
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 {
00043 namespace func_detail
00044 {
00045
00052 template <typename Complex>
00053 struct SimplexAggregator
00054 {
00056     using SimplexSet = typename casc::SimplexSet<Complex>;
00057
00063     SimplexAggregator(SimplexSet* p) : pLevels(p) {}
00064
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
00100 template <typename Complex>
00101 struct StarHelper
00102 {
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
00130 template <typename Complex>
00131 struct ClosureHelper

```



```

00132 {
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
00160 template <typename Complex>
00161 struct GraphVisitor
00162 {
00164     std::ostream &fout;
00165
00171     GraphVisitor(std::ostream &os) : fout(os) {}
00172
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
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
00255     void visit(const Complex &, typename Complex::template SimplexID<Complex::topLevel>){}
00256 };
00257
00264 template <typename Complex, typename K>
00265 struct DotHelper {};
00266
00273 template <typename Complex, std::size_t k>
00274 struct DotHelper<Complex, std::integral_constant<std::size_t, k> >
00275 {
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
00301 template <typename Complex>
00302 struct DotHelper<Complex, std::integral_constant<std::size_t, Complex::topLevel> >
00303 {
00310     static void printlevel(std::ostream &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
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
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
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
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
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 template <typename Complex, typename Simplex>
00437 void getLink(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
00438 {
00439     using SimplexSet = typename casc::SimplexSet<Complex>;
00440     SimplexSet star;
00441     SimplexSet closure;
00442     SimplexSet closeStar;
00443     SimplexSet starClose;
00444     getStar(F, s, star);
00445     getClosure(F, star, closeStar);
00446
00447     getClosure(F, s, closure);
00448     getStar(F, closure, starClose);
00449     casc::set_difference(closeStar, starClose, dest);
00450 }
00451
00452 template <typename Complex>
00453 void writeDOT(const std::string &filename, Complex &F)
00454 {
00455     // TODO: Put back the const F (0)
00456     std::ofstream fout(filename);
00457     if (!fout.is_open())
00458     {
00459         std::cerr << "File '" << filename
00460                 << "' could not be written to." << std::endl;
00461         fout.close();
00462         exit(1);
00463     }
00464
00465     fout << "digraph {\n"
00466         << "node [shape = record,height = .1]\n"
00467         << "splines=line;\n"
00468         << "dpi=300;\n";
00469     auto v = func_detail::GraphVisitor<Complex>(fout);
00470     visit_BFS_up(v, F, F.get_simplex_up());
00471
00472     // List the simplices
00473     func_detail::DotHelper<Complex,
00474                         std::integral_constant<std::size_t, 0> >::printlevel(fout, F);
00475     fout << ";\n";
00476     fout.close();
00477 }
00478 // 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
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     namespace visitor_detail
00044     {
00045
00056         template <typename Visitor, typename Traits, typename Complex, typename K>
00057         struct BFS_Up_Node {};
00058
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             static constexpr auto level = k;
00072             using CurrSimplexID = typename Complex::template SimplexID<level>;
00073             using NextSimplexID = typename Complex::template SimplexID<level+1>;
00074             template <typename T> using Container = typename Traits::template Container<T>;
00075
00081             using BFS_Up_Node_Next = BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t,
00082             level+1> >;
00083
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
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             static constexpr auto level = Complex::topLevel;
00125             using CurrSimplexID = typename Complex::template SimplexID<level>;
00126
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
00158         template <typename Visitor, typename Traits, typename Complex, typename K>
00159         struct BFS_Down_Node {};
00160
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             static constexpr auto level = k;
00174             using CurrSimplexID = typename Complex::template SimplexID<level>;
00175             using NextSimplexID = typename Complex::template SimplexID<level-1>;
00176             template <typename T> using Container = typename Traits::template Container<T>;
00177
00183             using BFS_Down_Node_Next = BFS_Down_Node<Visitor, Traits, Complex,
00184             std::integral_constant<std::size_t, level-1> >;

```

```

00184
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
00223 template <typename Visitor, typename Traits, typename Complex>
00224 struct BFS_Down_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, 1> >
00225 {
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
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
00272 template <typename Visitor, typename Traits, typename Complex, typename K>
00273 struct BFS_Edge {};
00274
00275
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     static constexpr auto level = k;
00288     using CurrEdgeID = typename Complex::template EdgeID<level>;
00289     using NextEdgeID = typename Complex::template EdgeID<level+1>;
00290     using CurrSimplexID = typename Complex::template SimplexID<level>;
00291     template <typename T> using Container = typename Traits::template Container<T>;
00292     using BFS_Edge_Next = BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t,
00293 level+1> >;
00294
00295     template <typename Iterator>
00296     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
00297     {
00298         Container<NextEdgeID> next;
00299         std::vector<typename Complex::KeyType> cover;
00300
00301         for (auto curr = begin; curr != end; ++curr)
00302         {
00303             v.visit(F, *curr);
00304
00305             CurrSimplexID n = curr->up();
00306             F.get_cover(n, std::back_inserter(cover));
00307             for (auto a : cover)
00308             {
00309                 NextEdgeID id = F.get_edge_up(n, a);
00310                 next.insert(next.end(), id);
00311             }
00312             cover.clear();
00313         }
00314
00315         BFS_Edge_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
00316     }
00317 };
00318
00322 template <typename Visitor, typename Traits, typename Complex>
00323 struct BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t, Complex::topLevel> >

```

```

00344 {
00346     static constexpr auto level = Complex::topLevel;
00348     using CurrEdgeID = typename Complex::template EdgeID<level>;
00349
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
00371 struct BFS_Repeat_Node_traits
00372 {
00374     template <typename T> using Container = std::vector<T>;
00375 };
00376
00378 struct BFS_NoRepeat_Node_Traits
00379 {
00381     template <typename T> using Container = NodeSet<T>;
00382 };
00383
00385 struct BFS_NoRepeat_Edge_Traits
00386 {
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
00394
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
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
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
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
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
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
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
00570 template <class Complex, std::size_t level, typename Iterator>
00571 void kneighbors_up(Complex &F, int ring, std::set<typename Complex::template SimplexID<level> > &nbors,
00572                  Iterator begin, Iterator end)
00573 {
00574     if (ring == 0)
00575     {
00576         return;
00577     }
00578     std::set<typename Complex::template SimplexID<level> > next;
00579     for (auto nid = begin; nid != end; ++nid)
00580     {
00581         for (auto a : F.get_cover(*nid))
00582         {
00583             auto id = F.get_simplex_up(*nid, a);
00584             for (auto b : F.get_name(id))
00585             {
00586                 auto nbor = F.get_simplex_down(id, b);
00587                 if (nbors.insert(nbor).second)
00588                 {
00589                     next.insert(nbor);
00590                 }
00591             }
00592         }
00593     }
00594     return kneighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
00595 }
00596
00597
00598
00599
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
00637 template <class Complex, std::size_t level, typename Iterator>
00638 void kneighbors(Complex &F, int ring, std::set<typename Complex::template SimplexID<level> > &nbors,
00639               Iterator begin, Iterator end)
00640 {
00641     if (ring == 0)
00642     {
00643         return;
00644     }
00645     std::set<typename Complex::template SimplexID<level> > next;
00646     for (auto nid = begin; nid != end; ++nid)
00647     {
00648         for (auto a : F.get_name(*nid))
00649         {
00650             auto id = F.get_simplex_down(*nid, a);
00651             for (auto b : F.get_cover(id))
00652             {
00653                 auto nbor = F.get_simplex_up(id, b);
00654                 if (nbors.insert(nbor).second)
00655                 {
00656                     next.insert(nbor);
00657                 }
00658             }
00659         }
00660     }

```



```

00661     }
00662     }
00663 }
00664 return neighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
00665 }
00666
00679 template <class Complex, class SimplexID>
00680 void neighbors(Complex &F, SimplexID nid, int ring, std::set<SimplexID> &nbors)
00681 {
00682     nbors.insert(nid);
00683     std::set<SimplexID> next {
00684         nid
00685     };
00686     neighbors<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 //     using Neighbors_Up_Node_Next =
00702 //     Neighbors_Up_Node<Visitor, Complex, level, ring-1>;
00703 //     template <typename Iterator>
00704 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00705 //     Iterator begin, Iterator end)
00706 //     {
00707 //         NodeSet<SimplexID> next;
00708 //         for(auto curr = begin; curr != end; ++curr)
00709 //         {
00710 //             if(v.visit(F, *curr))
00711 //             {
00712 //                 for(auto a : F.get_cover(*curr))
00713 //                 {
00714 //                     auto id = F.get_simplex_up(*curr, a);
00715 //                     for(auto b : F.get_name(id))
00716 //                     {
00717 //                         auto nbor = F.get_simplex_down(id, b);
00718 //                         if(nodes.insert(nbor).second)
00719 //                         {
00720 //                             next.insert(nbor);
00721 //                         }
00722 //                     }
00723 //                 }
00724 //             }
00725 //         }
00726 //     }
00727 // };
00728 //
00729 // Neighbors_Up_Node_Next::apply(std::forward<Visitor>(v), F, nodes,
00730 // next.begin(), next.end());
00731 // }
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 //     template <typename Iterator>
00741 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
00742 //     Iterator begin, Iterator end)
00743 //     {
00744 //         for(auto curr = begin; curr != end; ++curr)
00745 //         {
00746 //             v.visit(F, *curr);
00747 //         }
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 //     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 > &simplexMap)
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
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 {
00049 namespace decimation_detail
00050 {
00056 template <typename Complex>
00057 struct SimplexDataSet
00058 {
00060     using KeyType = typename Complex::KeyType;
00061
00068     template <std::size_t k, typename T>
00069     struct DataType
00070     {
00072         using type = std::pair<std::array<KeyType, k>, T>;
00073     };
00074
00080     template <std::size_t k>
00081     struct DataType<k, void>
00082     {
00084         using type = std::array<KeyType, k>;
00085     };
00086
00088     template <std::size_t j>
00089     using DataSet = typename DataType<j, typename Complex::template NodeData<j> >::type;
00091     using LevelIndex = typename std::make_index_sequence<Complex::numLevels>;
00093     using SimplexIDLevel = typename util::int_type_map<std::size_t,
00094                                                         std::tuple, LevelIndex, DataSet>::type;
00096     template <class T> using vector = std::vector<T>;
00098     using type = typename util::type_map<SimplexIDLevel, vector>::type;
00099 };
00100
00107 template <typename Complex>
00108 struct GetCompleteNeighborhood
00109 {
00111     using SimplexSet = typename casc::SimplexSet<Complex>;
00112
00118     GetCompleteNeighborhood(SimplexSet* p) : pLevels(p) {}
00119
00125     template <std::size_t level>
00126     bool visit(Complex &, typename Complex::template SimplexID<level>)
00127     {
00128         return true;
00129     }
00130
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:
00148         SimplexSet* pLevels;
00149 };
00150
00156 template <typename Complex>
00157 struct GrabVisitor
00158 {
00160     using SimplexSet = typename casc::SimplexSet<Complex>;
00161
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* pLevels;
00188     SimplexSet* pGrab;
00189 };
00190
00191 template <typename Complex, std::size_t BaseLevel>
00192 struct InnerVisitor
00193 {
00194     using SimplexSet = typename casc::SimplexSet<Complex>;
00195     using SimplexMap = typename casc::SimplexMap<Complex>;
00196     using Simplex = typename Complex::template SimplexID<BaseLevel>;
00197     using KeyType = typename Complex::KeyType;
00198
00199     InnerVisitor(SimplexSet* p, Simplex s, KeyType np, SimplexMap* rv)
00200     : pLevels(p), simplex(s), new_point(np), data(rv) {}
00201
00202     template <std::size_t OldLevel>
00203     bool visit(Complex &F, typename Complex::template SimplexID<OldLevel> s)
00204     {
00205         constexpr std::size_t NewLevel = OldLevel - BaseLevel + 1;
00206
00207         if (pLevels->find(s) != pLevels->template end<OldLevel>())
00208         {
00209             //std::cout << "InnerVisitor (found): " << s << std::endl;
00210             std::array<KeyType, OldLevel> old_name = F.get_name(s);
00211             std::array<KeyType, BaseLevel> base_name = F.get_name(simplex);
00212             using NewArrayType = std::array<KeyType, NewLevel>;
00213             NewArrayType new_name;
00214
00215             std::size_t i = 0;    // new_name
00216             std::size_t j = 0;    // old_name
00217             std::size_t k = 0;    // base_name
00218
00219             new_name[i++] = new_point;
00220
00221             // Remove base_name from old_name and append to new_name
00222             while (i < NewLevel)
00223             {
00224                 if (k >= BaseLevel) {
00225                     // append to new_name and increment
00226                     new_name[i++] = old_name[j++];
00227                     continue;
00228                 }
00229                 if (base_name[k] == old_name[j])
00230                 {
00231                     // if equivalent then skip the value
00232                     ++j; ++k;
00233                 }
00234                 else
00235                 {
00236                     // append to new_name and increment
00237                     new_name[i++] = old_name[j++];
00238                 }
00239             }
00240
00241             SimplexSet grab;
00242             visit_BFS_down(GrabVisitor<Complex>(pLevels, &grab), F, s);
00243
00244             auto &levelMap = casc::get<NewLevel>(*data);
00245             auto it = levelMap.find(new_name);
00246             if (it != levelMap.end())
00247             {
00248                 it->second.insert(grab);
00249             }
00250             else
00251             {
00252                 MAYBE_UNUSED auto ret = levelMap.insert(
00253                     std::pair<NewArrayType, SimplexSet>(new_name, grab));
00254                 assert(ret.second);
00255             }
00256         }
00257         return true;
00258     }
00259
00260 private:
00261     SimplexSet* pLevels;
00262     Simplex simplex;
00263     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         template <std::size_t k, typename ReturnType>
00318         struct PerformCallback
00319         {
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             {
00335                 static void apply(Complex &F, Callback<Complex> &&clbk,
00336                                     SimplexDataSet &rv,
00337                                     const std::array<KeyType, k> &new_name,
00338                                     const SimplexSet &merged)
00339                 {
00340                     clbk(F, new_name, merged);
00341                     std::get<k>(rv).push_back(new_name);
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
00410
00411
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
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
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
00485 template <typename Complex, typename Simplex, template<typename> class Callback>
00486 void decimate(Complex &F, Simplex s, Callback<Complex> &&clbk)
00487 {
00488     using SimplexSet = typename casc::SimplexSet<Complex>;
00489     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
00526 template <typename Complex, typename Simplex>
00527 typename Complex::KeyType decimateFirstHalf(Complex &F, Simplex s, SimplexMap<Complex> &simplexMap)
00528 {
00529     using SimplexSet = typename casc::SimplexSet<Complex>;
00530
00531     // Create the vertex to replace `s`
00532     typename Complex::KeyType np = F.add_vertex();
00533     SimplexSet nbhd;
00534
00535     // Get the complete neighborhood
00536     visit_BFS_down(
00537         decimation_detail::GetCompleteNeighborhood<Complex>(&nbhd),
00538         F, s);
00539
00540     // Call MainVisitor -> InnerVisitor -> GrabVisitor sequence
00541     visit_BFS_down(
00542         decimation_detail::MainVisitor<Complex>(
00543             &nbhd, np, &simplexMap),
00544         F, s);
00545     return np;
00546 }
00547
00559 template <typename Complex>
00560 void decimateBackHalf(Complex &F, SimplexMap<Complex> &simplexMap, typename
    decimation_detail::SimplexDataSet<Complex>::type &rv){
00561     SimplexSet<Complex> doomed;
00562     util::int_for_each<std::size_t, typename
    SimplexMap<Complex>::cLevelIndex>(decimation_detail::DoomedHelper<Complex>(), doomed, simplexMap);
00563
00564     perform_removal(F, doomed); // Remove simplices in the neighborhood
00565     perform_insertion(F, rv); // Insert new simplices
00566 }
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

```

```

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
00041 namespace index_tracker
00042 {
00044 namespace index_tracker_detail {
00045
00046
00052     template <typename T>
00053     struct Interval
00054     {
00056         Interval() : _a(0), _b(0) {}
00058         Interval(T a) : _a(a), _b(a+1) {}
00060         Interval(T a, T b) : _a(a), _b(b) { assert(a <= b); }
00062         Interval(const Interval<T>& rhs) : _a(rhs._a), _b(rhs._b) {}
00063
00071         Interval& operator=(const Interval& rhs)
00072         {
00073             _a = rhs._a;
00074             _b = rhs._b;
00075             return *this;
00076         }
00077
00079         bool has(T x) { return _a <= x && x < _b; }
00080
00082         T lower() const { return _a; }
00084         T upper() const { return _b; }
00085
00087         T& lower() { return _a; }
00089         T& upper() { return _b; }
00090
00092         std::size_t size() { return _b - _a; }
00093
00094     private:
00095         T _a;
00096         T _b;
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
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 template <typename Node>
00518 void fill_left(Pointer<Node> head, Data<Node>& x)
00519 {
00520     if(head->next[0] == nullptr)
00521     {
00522         Data<Node>& left = head->data[head->k-1];
00523         if(left.upper() == x.lower())
00524         {
00525             x.lower() = left.lower();
00526             --(head->k);
00527         }
00528     }
00529     else
00530     {
00531         fill_left<Node>(head->next[head->k], x);
00532         rebalance<Node>(head, head->k);
00533     }
00534 }
00535
00536 template <typename Node>
00537 void fill_right(Pointer<Node> head, Data<Node>& x)
00538 {
00539     if(head->next[0] == nullptr)
00540     {
00541         Data<Node>& right = head->data[0];
00542         if(right.lower() == x.upper())
00543         {
00544             x.upper() = right.upper();
00545             --(head->k);
00546             for(std::size_t i = 0; i < head->k; ++i)
00547             {
00548                 head->data[i] = head->data[i+1];
00549             }
00550         }
00551     }
00552     else
00553     {
00554         fill_right<Node>(head->next[0], x);
00555         rebalance<Node>(head, 0);
00556     }
00557 }
00558
00559 template <typename Node>
00560 void insert_scalar_H(Pointer<Node> head, Scalar<Node> data)
00561 {
00562     // If the
00563     if(head->next[0] == nullptr)
00564     {
00565         const auto k = head->k;
00566
00567         std::size_t i;
00568         for(i = 0; i < k; ++i)
00569         {
00570             Data<Node>& A = head->data[i];
00571             Scalar<Node> x = data;
00572
00573             if(x + 1 < A.lower())
00574             {
00575                 for(std::size_t j = k; j > i; --j)
00576                 {
00577                     head->data[j] = head->data[j-1];
00578                 }
00579             }
00580         }
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 template <typename Node>
00872 Data<Node> check_order(Pointer<Node> head, Data<Node> curr)
00873 {
00874     if(head != nullptr)
00875     {
00876         if(head->next[0] == nullptr)
00877         {
00878             for(std::size_t i = 0; i < head->k; ++i)
00879             {
00880                 if(curr > head->data[i])
00881                 {
00882                     std::cout << "ORDER WRONG!!!  --  " << curr << " > " << head->data[i] <<
00883 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] <<
00897 std::endl;
00898                     exit(1);
00899                 }
00900                 curr = head->data[i];
00901             }
00902             curr = check_order<Node>(head->next[head->k], curr);
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
00937 template <typename _T, std::size_t _d = 16>
00938 class index_tracker
00939 {
00940 public:
00942     using Node = index_tracker_detail::BTreeNode<_T, _d>;
00943     using T = _T;
00944     constexpr static std::size_t d = _d;
00945
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
00031 #pragma once
00032
00033 #include <iostream>
00034 #include <queue>
00035 #include <set>
00036
00037 namespace casc{
00042 struct Orientable {
00044     int orientation;
00045 };
00046
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 template <typename Complex>
00085 struct init_orientation_helper<Complex, std::integral_constant<std::size_t, Complex::topLevel>
00086 {
00087     static void f(Complex&) {}
00088 };
00089 // end namespace orientation_detail
00090
00091 template <typename Complex>
00092 void init_orientation(Complex& F)
00093 {
00094     orientation_detail::init_orientation_helper<Complex, std::integral_constant<std::size_t, 0>::f(F);
00095 }
00096
00097 template <typename Complex>
00098 void clear_orientation(Complex& F)
00099 {
00100     // clear orientation
00101     for(auto& curr : F.template get_level<Complex::topLevel>())
00102     {
00103         curr.orientation = 0;
00104     }
00105 }
00106
00107 // TODO: Implement this as a disjoint set operation during insertion (2)
00108 template <typename Complex>
00109 std::tuple<int, bool, bool> compute_orientation(Complex& F)
00110 {
00111     init_orientation(F);
00112     clear_orientation(F);
00113     return check_orientation(F);
00114 }
00115
00116 template <typename Complex>
00117 std::tuple<int, bool, bool> check_orientation(Complex& F)
00118 {
00119     // compute orientation
00120     constexpr std::size_t k = Complex::topLevel - 1;
00121
00122     std::deque<typename Complex::template SimplexID<k>> frontier;
00123     std::set<typename Complex::template SimplexID<k>> visited;
00124     int connected_components = 0;
00125     bool orientable = true;
00126     bool psuedo_manifold = true;
00127     for(auto outer : F.template get_level_id<k>())
00128     {
00129         if(visited.find(outer) == visited.end())
00130         {
00131             ++connected_components;
00132             frontier.push_back(outer);
00133
00134             while(!frontier.empty())
00135             {
00136                 typename Complex::template SimplexID<k> curr = frontier.front();
00137                 if(visited.find(curr) == visited.end())
00138                 {
00139                     visited.insert(curr);
00140
00141                     auto w = F.get_cover(curr);
00142
00143                     if(w.size() == 1)
00144                     {
00145                         // w is a boundary
00146                         //std::cout << curr << ":" << w[0] << " ~ Boundary" << std::endl;
00147                     }
00148                     else if(w.size() == 2)
00149                     {
00150                         auto& edge0 = *F.get_edge_up(curr, w[0]);
00151                         auto& edge1 = *F.get_edge_up(curr, w[1]);
00152
00153                         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
00030 #pragma once
00031
00032 #include <array>
00033 #include <map>
00034 #include "util.h"
00035 #include "stringutil.h"
00036
00037 namespace casc
00038 {
00039
00046 template <typename Complex>
00047 struct SimplexMap
00048 {
00050     template <std::size_t j>
00051     using SimplexID = typename Complex::template SimplexID<j>;
00053     using LevelIndex = typename Complex::LevelIndex;
00055     using cLevelIndex = typename util::remove_first_val<std::size_t,
00056                                                         LevelIndex>::type;
00058     using RevIndex = typename util::reverse_sequence<std::size_t,
00059                                                         LevelIndex>::type;

```

```

00061     using cRevIndex = typename util::reverse_sequence<std::size_t,
00062                                                         cLevelIndex>::type;
00064     using type_this = SimplexMap<Complex>;
00065
00066     SimplexMap() {};
00070
00071     // TODO: Put in convenience functions for easy accession etc... (0)
00079     template <std::size_t k>
00080     inline auto &get()
00081     {
00082         return std::get<k>(tupleMap);
00083     }
00084
00088     template <std::size_t k>
00089     inline auto &get() const
00090     {
00091         return std::get<k>(tupleMap);
00092     }
00093
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:
00115         struct PrintHelper
00116         {
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
00139         template <std::size_t k> using array = std::array<typename Complex::KeyType, k>;
00141         using ArrayLevel = typename
00142         util::int_type_map<std::size_t, std::tuple, LevelIndex, array>::type;
00143         template <class T> using map = std::map<T, SimplexSet<Complex> >;
00145         typename util::type_map<ArrayLevel, map>::type tupleMap;
00146     };
00147
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
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
00030 #pragma once
00031
00032 #include <algorithm>
00033 #include <unordered_set>
```

```

00034 #include "util.h"
00035
00036 namespace casc
00037 {
00038
00048 template <typename Complex>
00049 struct SimplexSet
00050 {
00052     template <std::size_t j>
00053     using SimplexID = typename Complex::template SimplexID<j>;
00055     using LevelIndex = typename Complex::LevelIndex;
00057     using cLevelIndex = typename util::remove_first_val<std::size_t,
00058                                                         LevelIndex>::type;
00060     using RevIndex = typename util::reverse_sequence<std::size_t,
00061                                                         LevelIndex>::type;
00063     using cRevIndex = typename util::reverse_sequence<std::size_t,
00064                                                         cLevelIndex>::type;
00066     using type_this = SimplexSet<Complex>;
00067
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.
00074     typename util::type_map<SimplexIDLevel, NodeSet>::type tupleSet;
00075
00077     SimplexSet() {};
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
00096     template <std::size_t k>
00097     inline auto empty() const noexcept{
00098         return std::get<k>(tupleSet).empty();
00099     }
00100
00109     template <std::size_t k>
00110     inline auto size() const noexcept{
00111         return std::get<k>(tupleSet).size();
00112     }
00113
00117     void clear()
00118     {
00119         util::int_for_each<std::size_t, LevelIndex>(ClearHelper(), this);
00120     }
00121
00129     template <std::size_t k>
00130     inline void insert(SimplexID<k> s)
00131     {
00132         std::get<k>(tupleSet).insert(s);
00133     }
00134
00140     void insert(const SimplexSet<Complex> &s)
00141     {
00142         util::int_for_each<std::size_t, LevelIndex>(
00143             InsertHelper(), this, s);
00144     }
00145
00153     template <std::size_t k>
00154     inline void erase(SimplexID<k> s)
00155     {
00156         std::get<k>(tupleSet).erase(s);
00157     }
00158
00164     void erase(const SimplexSet<Complex> &s)
00165     {
00166         util::int_for_each<std::size_t, LevelIndex>(
00167             EraseHelper(), this, s);
00168     }
00169
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
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

```

```

00212     template <std::size_t k>
00213     inline auto end()
00214     {
00215         return std::get<k>(tupleSet).end();
00216     }
00217
00226     template <std::size_t k>
00227     inline auto cend() const
00228     {
00229         return std::get<k>(tupleSet).cend();
00230     }
00231
00239     template <std::size_t k>
00240     inline auto begin()
00241     {
00242         return std::get<k>(tupleSet).begin();
00243     }
00244
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
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:
00308     struct InsertHelper
00309     {
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
00332     struct EraseHelper
00333     {
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

```

```

00356     struct PrintHelper
00357     {
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
00382     struct ClearHelper
00383     {
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
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
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
00443 namespace simplex_set_detail
00444 {
00445
00451 template <typename Complex>
00452 struct UnionH
00453 {
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
00483 template <typename Complex>
00484 struct IntersectH
00485 {
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
00530 template <typename Complex>
00531 struct DifferenceH
00532 {
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
00566 template <typename Complex>
00567 struct OperatorEQH
00568 {
00570     bool result;
00571
00573     OperatorEQH(): result(true) {}
00574
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
00593
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
00622 template <typename Complex>
00623 bool operator!=(const SimplexSet<Complex> &lhs, const SimplexSet<Complex> &rhs){
00624     return !(lhs == rhs);
00625 }
00626
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
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
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](#)
- template<typename T>
using [casc::NodeSet](#)
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
00031 #pragma once
00032
00033 #include <algorithm>
00034 #include <assert.h>
00035 #include <stdint>
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
00061 namespace casc
00062 {
00065 namespace detail
00066 {
00068 template <class T> using map = std::map<std::size_t, T>;
00069
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
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
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 template <class KeyType, std::size_t k, std::size_t N, typename DataTypes, class> struct asc_Node;
00283
00284 struct asc_NodeBase {
00285     asc_NodeBase(std::size_t id) : _node(id) {}

```

```

00299     virtual ~asc_NodeBase() {};
00300     std::size_t _node;
00301 };
00302
00308 template <class DataType>
00309 struct asc_NodeData {
00310     DataType _data;
00311 };
00312
00319 template <>
00320 struct asc_NodeData<void> {};
00321
00328 template <class KeyType, class DataType>
00329 struct asc_EdgeData {
00331     std::unordered_map<KeyType, DataType> _edge_data;
00332 };
00333
00339 template <class KeyType>
00340 struct asc_EdgeData<KeyType, void> {};
00341
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> {
00360     using DownNodeT = asc_Node<KeyType, k-1, N, NodeDataTypes, EdgeDataTypes>;
00361
00363     asc_arraymap<KeyType, DownNodeT*, k> _down;
00364     // std::map<KeyType, DownNodeT*> _down;
00365 };
00366
00376 template < class KeyType,
00377             std::size_t k,
00378             std::size_t N,
00379             class NodeDataTypes,
00380             class EdgeDataTypes>
00381 struct asc_NodeUp {
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
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 {
00404     static constexpr std::size_t level = k;
00405
00411     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00412
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
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 {

```

```

00462     static constexpr std::size_t level = 0;
00463
00469     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00470
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
00505 template <class KeyType, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
00506 struct asc_Node<KeyType, N, NodeDataTypes, EdgeDataTypes> :
00507     public asc_NodeBase,
00508     public asc_NodeData<typename util::type_get<N, NodeDataTypes>::type>,
00509     public asc_NodeDown<KeyType, N, NodeDataTypes, EdgeDataTypes>
00510 {
00511     static constexpr std::size_t level = N;
00512
00513
00519     asc_Node(std::size_t id) : asc_NodeBase(id) {}
00520
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
00553 template <typename Iter, typename Data>
00554 struct node_id_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
00555     public:
00557         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
00559         node_id_iterator() {}
00561         node_id_iterator(Iter j) : i(j) {}
00563         node_id_iterator &operator++() { ++i; return *this; }
00565         node_id_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
00567         node_id_iterator &operator--() { --i; return *this; }
00569         node_id_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
00571         bool operator==(node_id_iterator j) const { return i == j.i; }
00573         bool operator!=(node_id_iterator j) const { return !(*this == j); }
00575         Data operator*() { return Data(i->second); }
00577         const Data operator*() const { return Data(i->second); }
00579         typename super::pointer operator->() { return Data(i->second); }
00580     protected:
00582         Iter i;
00583 };
00584
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
00607 template <typename Iter, typename Data>
00608 struct node_data_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
00609     public:
00611         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
00613         node_data_iterator() {}
00615         node_data_iterator(Iter j) : i(j) {}
00617         node_data_iterator &operator++() { ++i; return *this; }
00619         node_data_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
00621         node_data_iterator &operator--() { --i; return *this; }
00623         node_data_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
00625         bool operator==(node_data_iterator j) const { return i == j.i; }
00627         bool operator!=(node_data_iterator j) const { return !(*this == j); }

```

```

00629         typename super::reference operator*() { return i->second->_data; }
00631         typename super::pointer operator->() { return i->second->_data; }
00632     protected:
00633         Iter i;
00634     };
00635 };
00636
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
00661 template <typename K, typename ... Ts>
00662 struct simplicial_complex_traits_default
00663 {
00664     template <std::size_t k> using all_int = int;
00665     using KeyType = K;
00666     using NodeTypes = util::type_holder<Ts...>;
00667     using EdgeTypes = typename util::int_type_map<std::size_t,
00668                                                 util::type_holder,
00669                                                 typename std::make_index_sequence<sizeof ...
00670     (Ts)-1>,
00671                                                 all_int>::type;
00672 };
00673 // end namespace detail
00674
00702 template <typename traits>
00703 class simplicial_complex
00704 {
00705     public:
00706         using KeyType = typename traits::KeyType;
00707         using NodeDataTypes = typename traits::NodeTypes;
00708         using EdgeDataTypes = typename traits::EdgeTypes;
00709         using type_this = simplicial_complex<traits>;
00710         static constexpr std::size_t numLevels = NodeDataTypes::size;
00711         static constexpr std::size_t topLevel = numLevels-1;
00712         static constexpr std::size_t bdryLevel = numLevels-2;
00713         using LevelIndex = typename std::make_index_sequence<numLevels>;
00714     private:
00715         template <std::size_t k> using Node = detail::asc_Node<KeyType, k, topLevel, NodeDataTypes,
00716         EdgeDataTypes>;
00717         template <std::size_t k> using NodePtr = Node<k>*>;
00718     public:
00719         template <std::size_t k> using NodeData = typename util::type_get<k, NodeDataTypes>::type;
00720         template <std::size_t k> using EdgeData = typename util::type_get<k, EdgeDataTypes>::type;
00721
00722         friend struct SimplexID;
00723         template <std::size_t k>
00724         struct SimplexID {
00725             using complex = simplicial_complex<traits>;
00726             friend simplicial_complex<traits>;
00727             static constexpr std::size_t level = k;
00728
00729             SimplexID() : ptr(nullptr) {}
00730
00731             SimplexID(NodePtr<k> p) : ptr(p) {}
00732
00733             SimplexID(const SimplexID &rhs) : ptr(rhs.ptr) {}
00734
00735             SimplexID &operator=(const SimplexID &rhs) { ptr = rhs.ptr; return *this; }
00736
00737             friend bool operator==(SimplexID lhs, SimplexID rhs) { return lhs.ptr == rhs.ptr; }
00738             friend bool operator!=(SimplexID lhs, SimplexID rhs) { return lhs.ptr != rhs.ptr; }
00739             friend bool operator<=(SimplexID lhs, SimplexID rhs) { return lhs.ptr <= rhs.ptr; }
00740             friend bool operator>=(SimplexID lhs, SimplexID rhs) { return lhs.ptr >= rhs.ptr; }
00741             friend bool operator<(SimplexID lhs, SimplexID rhs) { return lhs.ptr < rhs.ptr; }
00742             friend bool operator>(SimplexID lhs, SimplexID rhs) { return lhs.ptr > rhs.ptr; }
00743
00744             explicit operator std::uintptr_t () const { return reinterpret_cast<std::uintptr_t>(ptr); }
00745         };
00746
00747         complex::NodeData<k> const &operator*() const { return ptr->_data; }
00748         complex::NodeData<k> &operator*() { return ptr->_data; }
00749
00750         complex::NodeData<k> const &data() const { return ptr->_data; }
00751         complex::NodeData<k> &data() { return ptr->_data; }
00752
00753         std::array<KeyType, k> indices() const
00754         {
00755             std::array<KeyType, k> s;
00756             std::size_t i = 0;
00757             for (auto curr : ptr->_down)
00758             {
00759                 s[i++] = curr.first;
00760             }
00761         }
00762     };

```

```

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){
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
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
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
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
00898     SimplexID<k+1> get_simplex_up(const KeyType s) const
00899     {
00900         return get_recurse<k, 1>::apply(&s, this->ptr);
00901     }
00902
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
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
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
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:
00992         template <std::size_t l, std::size_t foo>
00993         struct print_helper
00994         {
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
01023         template <std::size_t foo>
01024         struct print_helper<0, foo>
01025         {
01034             static std::ostream &apply(std::ostream &out,
01035                                         const SimplexID &nid)
01036             {
01037                 out << "root " << nid;
01038                 return out;
01039             }
01040         };
01042         NodePtr<k> ptr;
01043     };
01044
01045     friend struct EdgeID;
01054     template <std::size_t k>
01055     struct EdgeID {
01057         using complex = simplicial_complex<traits>;
01059         friend simplicial_complex<traits>;
01061         static constexpr std::size_t level = k;
01062
01066         EdgeID() : ptr(nullptr), edge(0) {}
01067
01074         EdgeID(NodePtr<k> p, KeyType e) : ptr(p), edge(e) {}
01075
01081         EdgeID(const EdgeID &rhs) : ptr(rhs.ptr), edge(rhs.edge) {}
01082
01084         EdgeID &operator=(const EdgeID &rhs) { ptr = rhs.ptr; edge = rhs.edge; return *this; }
01085
01087         friend bool operator==(EdgeID lhs, EdgeID rhs) { return lhs.ptr == rhs.ptr && lhs.edge ==
rhs.edge; }
01089         friend bool operator!=(EdgeID lhs, EdgeID rhs) { return !(lhs == rhs); }
01091         friend bool operator<=(EdgeID lhs, EdgeID rhs) { return lhs < rhs || lhs == rhs; }
01093         friend bool operator>=(EdgeID lhs, EdgeID rhs) { return lhs > rhs || lhs == rhs; }
01095         friend bool operator<(EdgeID lhs, EdgeID rhs)
01096         {
01097             return (lhs.ptr < rhs.ptr) || (lhs.ptr == rhs.ptr && lhs.edge < rhs.edge);
01098         }
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
01106         auto const &operator*() const { return data(); }
01108         auto &operator*() { return data(); }
01109
01111         KeyType      key() const { return edge; }
01112
01114         auto const &data() const { return ptr->_edge_data[edge]; }
01116         auto          &data() { return ptr->_edge_data[edge]; }
01117
01123         SimplexID<k> up() const { return ptr; }
01124
01130         SimplexID<k-1> down() const { return SimplexID<k-1>(ptr->_down[edge]); }
01131
01132     private:
01134         NodePtr<k> ptr;
01136         KeyType      edge;
01137     };
01138

```



```

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
01161     ~simplicial_complex()
01162     {
01163         std::size_t count;
01164         remove_recurse<0, 0>::apply(this, &_root, &_root + 1, count);
01165     }
01166
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
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
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
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
01263     KeyType add_vertex()
01264     {
01265         KeyType v[1] = {unused_vertices.pop()};
01266         insert<1>(v);
01267         return v[0];
01268     }
01269
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
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
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     std::array<KeyType, 0> get_name(SimplexID<0>) const
01324     {
01325         std::array<KeyType, 0> name{};
01326         return name;
01327     }
01328
01329
01330     template <std::size_t n>
01331     SimplexID<n> get_simplex_up(const KeyType (&s)[n]) const
01332     {
01333         return get_recurse<0, n>::apply(s, _root);
01334     }
01335
01336     template <std::size_t n>
01337     SimplexID<n> get_simplex_up(const std::array<KeyType, n> &arr) const
01338     {
01339         return get_recurse<0, n>::apply(arr.data(), _root);
01340     }
01341
01342     template <std::size_t i, std::size_t j>
01343     SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const KeyType (&s)[j]) const
01344     {
01345         return get_recurse<i, j>::apply(s, id);
01346     }
01347
01348     template <std::size_t i, std::size_t j>
01349     SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const std::array<KeyType, j> &arr) const
01350     {
01351         return get_recurse<i, j>::apply(arr.data(), id);
01352     }
01353
01354     template <std::size_t i>
01355     SimplexID<i+1> get_simplex_up(const SimplexID<i> id, const KeyType s) const
01356     {
01357         return get_recurse<i, 1>::apply(&s, id.ptr);
01358     }
01359
01360     SimplexID<0> get_simplex_up() const
01361     {
01362         return _root;
01363     }
01364
01365     template <std::size_t i, std::size_t j>
01366     SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const KeyType (&s)[j]) const
01367     {
01368         return get_down_recurse<i, j>::apply(s, id.ptr);
01369     }
01370
01371     template <std::size_t i, std::size_t j>
01372     SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const std::array<KeyType, j> &arr)
01373     const
01374     {
01375         return get_down_recurse<i, j>::apply(arr.data(), id.ptr);
01376     }
01377
01378     template <std::size_t i>
01379     SimplexID<i-1> get_simplex_down(const SimplexID<i> id, const KeyType s) const
01380     {
01381         return get_down_recurse<i, 1>::apply(&s, id.ptr);
01382     }
01383
01384     SimplexID<0> get_simplex_down() const
01385     {
01386         return _root;
01387     }
01388
01389     template <std::size_t k, class Inserter>
01390     void get_cover_insert(const SimplexID<k> id, Inserter pos) const
01391     {
01392         for (auto curr : id.ptr->_up)
01393         {
01394             *pos++ = curr.first;
01395         }
01396     }

```

```

01483     }
01484
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
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
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
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
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 template <std::size_t k>
01621 std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &&simplices) const
01622 {
01623     std::set<SimplexID<k-1> > rval;
01624     for (auto nid : simplices)
01625     {
01626         for (auto p : nid.ptr->_down)
01627         {
01628             rval.insert(SimplexID<k-1>(p.second));
01629         }
01630     }
01631     return rval;
01632 }
01633
01634 template <std::size_t k>
01635 std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &simplices) const
01636 {
01637     std::set<SimplexID<k-1> > rval;
01638     for (auto simplex : simplices)
01639     {
01640         for (auto p : simplex.ptr->_down)
01641         {
01642             rval.insert(SimplexID<k-1>(p.second));
01643         }
01644     }
01645     return rval;
01646 }
01647
01648 template <std::size_t k>
01649 std::set<SimplexID<k-1> > down(const SimplexID<k> simplex) const
01650 {
01651     std::set<SimplexID<k-1> > rval;
01652     for (auto p : simplex.ptr->_down)
01653     {
01654         rval.insert(SimplexID<k-1>(p.second));
01655     }
01656     return rval;
01657 }
01658
01659 template <std::size_t k, class InsertIter>
01660 void down(const std::set<SimplexID<k>>&& simplices, InsertIter iter) const{
01661     for (auto simplex : simplices)
01662     {
01663         for (auto p : simplex.ptr->_down)
01664         {
01665             *iter++ = SimplexID<k-1>(p.second);
01666         }
01667     }
01668 }
01669
01670 template <std::size_t k, class InsertIter>
01671 void down(const std::set<SimplexID<k>>& simplices, InsertIter iter) const{
01672     for (auto simplex : simplices)
01673     {
01674         for (auto p : simplex.ptr->_down)
01675         {
01676             *iter++ = SimplexID<k-1>(p.second);
01677         }
01678     }
01679 }
01680
01681 template <std::size_t k, class InsertIter>
01682 void down(const SimplexID<k> simplex, InsertIter iter) const{
01683     for (auto p : simplex.ptr->_down)
01684     {
01685         *iter++ = SimplexID<k-1>(p.second);
01686     }
01687 }
01688
01689 template <std::size_t k>
01690 EdgeID<k+1> get_edge_up(SimplexID<k> simplex, KeyType a)
01691 {
01692     return EdgeID<k+1>(simplex.ptr->_up.at(a), a);
01693 }
01694
01695 template <std::size_t k>
01696 EdgeID<k> get_edge_down(SimplexID<k> simplex, KeyType a)
01697 {
01698     return EdgeID<k>(simplex.ptr, a);
01699 }
01700
01701 template <std::size_t k>
01702 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
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
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
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
01811     detail::map<NodePtr<k>>::iterator, SimplexID<k>;
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
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>
01843 >(begin);
01844         auto data_end = detail::make_node_id_iterator<decltype(end), const SimplexID<k> >(end);
01845         return util::make_range(data_begin, data_end);
01846     }
01847
01848     // template <std::size_t k> using DataIterator = detail::node_data_iterator<typename
01849     std::map<std::size_t, NodePtr<k>>::iterator, NodeData<k>;
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
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>
01883 >(begin);
01884         auto data_end = detail::make_node_data_iterator<decltype(end), const NodeData<k> >(end);
01885         return util::make_range(data_begin, data_end);
01886     }
01897
01898     template <std::size_t k>
01899     std::size_t remove(const KeyType (&s)[k])
01900     {
01901         Node<k>* root = get_recurse<0, k>::apply(s, _root);
01902         std::size_t count = 0;
01903         return remove_recurse<k, 0>::apply(this, &root, &root + 1, count);
01904     }
01914
01915     template <std::size_t k>
01916     std::size_t remove(const std::array<KeyType, k> &s)
01917     {
01918         Node<k>* root = get_recurse<0, k>::apply(s.data(), _root);
01919         std::size_t count = 0;
01920         return remove_recurse<k, 0>::apply(this, &root, &root + 1, count);
01921     }

```

```

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     template <std::size_t k>
01940     bool onBoundary(const SimplexID<k> s) const
01941     {
01942         return onBoundaryH<k, 0>::apply(s);
01943     }
01944
01945     template <std::size_t level>
01946     bool nearBoundary(const SimplexID<level> s) const
01947     {
01948         return nearBoundaryH<level, 0>::apply(s);
01949     }
01950
01951     /** Reintroduce this code block when this is resolved
01952     // http://www.open-std.org/jtc1/sc22/wg21/docs/cwg\_defects.html#727
01953
01954     // /**
01955     // * @brief      Checks whether a simplex is on a boundary.
01956     // *
01957     // * @param[in]   s      SimplexID of interest
01958     // *
01959     // * @tparam      k      Dimension of the simplex
01960     // *
01961     // * @return      True if the simplex interacts with a
01962     // *              topLevel-1 simplex which is on a boundary.
01963     // */
01964     // template <std::size_t k>
01965     // bool onBoundary(const SimplexID<k> s) const
01966     // {
01967     //     for(auto p : s.ptr->_up)
01968     //     {
01969     //         if(onBoundary(SimplexID<k+1>(p.second)))
01970     //             return true;
01971     //     }
01972     //     return false;
01973     // }
01974
01975     // /**
01976     // * @brief      Specialization of the facets
01977     // *
01978     // * @param[in]   s      SimplexID of interest
01979     // *
01980     // * @tparam      k      Dimension of the simplex
01981     // *
01982     // * @return      True if s is on a boundary
01983     // */
01984     // template<>
01985     // bool onBoundary(const SimplexID<topLevel> s) const
01986     // {
01987     //     for(auto p : s.ptr->_down){
01988     //         if(onBoundary(SimplexID<topLevel-1>(p.second)))
01989     //             return true;
01990     //     }
01991     //     return false;
01992     // }
01993
01994     // /**
01995     // * @brief      Specialization of the topLevel-1 simplices
01996     // *
01997     // * @param[in]   s      SimplexID of interest
01998     // *
01999     // * @tparam      k      Dimension of the simplex
02000     // *
02001     // * @return      True if s is on a boundary
02002     // */
02003     // template<>
02004     // bool onBoundary(const SimplexID<topLevel-1> s) const
02005     // {
02006     //     return s.ptr->_up.size() != 2;
02007     // }
02008
02009     template <std::size_t L, std::size_t R>
02010     bool leq(SimplexID<L> lhs, SimplexID<R> rhs) const
02011     {
02012         auto name_lhs = get_name(lhs);
02013         auto name_rhs = get_name(rhs);
02014
02015         std::size_t i = 0;
02016         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
02073     template <std::size_t L, std::size_t R>
02074     bool eq(SimplexID<L>, SimplexID<R>) const
02075     {
02076         return false;
02077     }
02078
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
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:
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
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
02176     template <std::size_t level, std::size_t foo>
02177     struct onBoundaryH
02178     {
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
02203     template <std::size_t foo>
02204     struct onBoundaryH<topLevel, foo>

```

```

02205     {
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
02229 template <std::size_t foo>
02230 struct onBoundaryH<bdryLevel, foo>
02231 {
02239     static bool apply(const SimplexID<bdryLevel> s)
02240     {
02241         return s.ptr->_up.size() < 2;
02242     }
02243 };
02244
02252 template <std::size_t level, std::size_t foo>
02253 struct remove_recurse
02254 {
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
02294 template <std::size_t foo>
02295 struct remove_recurse<topLevel, foo>
02296 {
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
02328 template <std::size_t level, std::size_t n>
02329 struct get_recurse
02330 {
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 };
02367 template <std::size_t level>
02368 struct get_recurse<level, 0>
02369 {
02379     static Node<level>* apply(const KeyType*, Node<level>* root)
02380     {

```



```

02381         return root;
02382     }
02383 };
02384
02391 template <std::size_t level, std::size_t n>
02392 struct get_down_recurse
02393 {
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
02429 template <std::size_t level>
02430 struct get_down_recurse<level, 0>
02431 {
02441     static Node<level>* apply(const KeyType*, Node<level>* root)
02442     {
02443         return root;
02444     }
02445 };
02446
02454 template <std::size_t level, std::size_t n>
02455 struct insert_full
02456 {
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
02479 template <std::size_t level>
02480 struct insert_full<level, 0>
02481 {
02491     static Node<level>* apply(type_this*, Node<level>* root, const KeyType*)
02492     {
02493         return root;
02494     }
02495 };
02496
02504 template <std::size_t level, std::size_t antistep, std::size_t n>
02505 struct insert_for
02506 {
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
02529 template <std::size_t level, std::size_t n>
02530 struct insert_for<level, 1, n>
02531 {
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
02553 template <std::size_t level, std::size_t n>
02554 struct insert_raw
02555 {
02565     static Node<level+n+1>* apply(type_this* that, Node<level>* root, const KeyType* begin)
02566     {
02567         KeyType v = *(begin+n);
02568         Node<level+1>* nn;
02569         // if root->v doesn't exist then create it
02570         auto iter = root->_up.find(v);
02571         if (iter == root->_up.end())
02572         {
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
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
02621 void backfill(Node<0>*, Node<1>*, int)
02622 {
02623     return;
02624 }
02625
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
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
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
02706     void remove_node(Node<0>* p)
02707     {
02708         for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
02709         {
02710             curr->second->_down.erase(curr->first);
02711         }
02712         --(level_count[0]);
02713         std::get<0>(levels).erase(p->_node);
02714         delete p;
02715     }
02716
02717     void remove_node(Node<topLevel>* p)
02723     {
02724         for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
02725         {
02726             curr->second->_up.erase(curr->first);
02727         }
02728         --(level_count[topLevel]);
02729         std::get<topLevel>(levels).erase(p->_node);
02730         delete p;
02731     }
02732
02733     NodePtr<0> _root;
02735     std::size_t node_count;
02737     std::array<std::size_t, numLevels> level_count;
02739     using NodePtrLevel = typename
02741     util::int_type_map<std::size_t, std::tuple, LevelIndex, NodePtr>::type;
02743     typename util::type_map<NodePtrLevel, detail::map>::type levels;
02745     index_tracker::index_tracker<KeyType> unused_vertices;
02746 };
02747
02748
02764 template <typename KeyType, typename ... Ts>
02765 using AbstractSimplicialComplex = simplicial_complex<
02766     detail::simplicial_complex_traits_default<KeyType, Ts...> >;
02767
02769 namespace simplex_set_detail{
02780 template <typename SimplexID>
02781 struct hashSimplexID{
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
02799
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
00030 #pragma once
00031
00032 #include <string>
00033
00034 namespace casc
00035 {
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
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
00085 template <class T>
00086 CONSTEXPR14_TN
00087 static_string
00088 type_name()
00089 {
00090     #ifdef __clang__
00091         static_string p = __PRETTY_FUNCTION__;
00092         return static_string(p.data() + 31, p.size() - 31 - 1);
00093     #elif defined(__GNUC__)
00094         static_string p = __PRETTY_FUNCTION__;
00095         # if __cplusplus < 201402
00096             return static_string(p.data() + 36, p.size() - 36 - 1);
00097         # else
00098             return static_string(p.data() + 46, p.size() - 46 - 1);
00099         # endif
00100     #elif defined(_MSC_VER)
00101         static_string p = __FUNCSIG__;
00102         return static_string(p.data() + 38, p.size() - 38 - 7);
00103     #endif
00104 }

```

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_i , 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
00030 #pragma once
00031
00032 #include <utility>
00033 #include <array>
00034
00036 namespace util
00037 {
00048 template<typename T> struct range
00049 {
```

```

00050
00058     template <class C>
00059     range(C &&c) : _begin(c.begin()), _end(c.end()) {}
00060
00067     range(T b, T e) : _begin(b), _end(e) {}
00068
00074     T begin() { return _begin; }
00075
00081     T end() { return _end; }
00082
00083     private:
00085         T _begin;
00087         T _end;
00088 };
00089
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
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
00127 template <typename ... Ts>
00128 struct type_holder
00129 {
00131     static const std::size_t size = sizeof ... (Ts);
00132 };
00133
00140 template <typename T, typename ... Ts>
00141 struct type_holder<T, Ts...>
00142 {
00144     using head = T;
00146     using tail = type_holder<Ts...>;
00148     static const std::size_t size = 1 + type_holder<Ts...>::size;
00149 };
00150
00159 template <std::size_t k, typename T>
00160 struct type_get {};
00161
00167 template <typename ... Ts>
00168 struct type_get<0, type_holder<Ts...> >
00169 {
00171     using type = typename type_holder<Ts...>::head;
00172 };
00173
00180 template <std::size_t k, typename ... Ts>
00181 struct type_get<k, type_holder<Ts...> >
00182 {
00184     using type = typename type_get<k-1, typename type_holder<Ts...>::tail>::type;
00185 };
00186
00189 namespace detail
00190 {
00198     template <class C, template <typename> class V, typename ... Rs>
00199     struct type_map_helper {};
00200
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
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
00230
00241 template <class C, template <typename> class V>
00242 struct type_map
00243 {
00245     using type = typename detail::type_map_helper<C, V>::type;
00246 };
00247
00249 namespace detail
00250 {
00254     template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
        <IntegerType> class F, typename ... Accumulators>
00255     struct int_type_map_helper {};
00256
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
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
00290
00302 template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
<IntegerType> class F>
00303 struct int_type_map
00304 {
00306     using type = typename detail::int_type_map_helper<IntegerType, OutHolder, IntegerSequence,
F>::type;
00307 };
00308
00315 template <template <class ...> class TUPLE, typename HOLDER_FULL>
00316 struct type_swap
00317 {};
00318
00326 template <template <class ...> class TUPLE, template <class ...> class HOLDER, typename ... Ts>
00327 struct type_swap<TUPLE, HOLDER<Ts...> >
00328 {
00330     using type = TUPLE<Ts...>;
00331 };
00332
00334 namespace detail
00335 {
00343 template <class Integer, class IntegerSequence, Integer... Accumulator>
00344 struct reverse_sequence_helper {};
00345
00353 template <class Integer,
00354     template<class, Integer...> class InHolder,
00355     Integer... Accumulator>
00356 struct reverse_sequence_helper<Integer, InHolder<Integer>, Accumulator...>
00357 {
00359     using type = InHolder<Integer, Accumulator...>;
00360 };
00361
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.
00379     using type = typename reverse_sequence_helper<Integer,
00380         InHolder<Integer, Is...>, I, Accumulator...>::type;
00381 };
00382 } // end namespace detail
00384
00391 template <class Integer, class IntegerSequence>
00392 struct reverse_sequence
00393 {
00395     using type = typename detail::reverse_sequence_helper<Integer, IntegerSequence>::type;
00396 };
00397
00398
00405 template <class Integer, class IntegerSequence>
00406 struct remove_first_val {};
00407
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 {
00423     using type = InHolder<Integer, Is...>;
00424 };
00425
00427 namespace detail
00428 {
00433 template <typename Integer, typename IntegerSequence, typename Fn, typename ... Args>
00434 struct int_for_each_helper {};
00435
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 template <class Integer, template <class, Integer...> class InHolder,
00456         Integer I, Integer... Is, typename Fn, typename ... Args>
00457 struct int_for_each_helper<Integer, InHolder<Integer, I, Is...>, Fn, Args...>
00458 {
00459     static void apply(Fn &&f, Args && ... args)
00460     {
00461         f.template apply<I>(std::forward<Args>(args) ...);
00462         int_for_each_helper<Integer, InHolder<Integer, Is...>, Fn, Args...>::apply(
00463             std::forward<Fn>(f),
00464             std::forward<Args>(args) ...);
00465     }
00466 };
00467 } // end namespace detail
00468
00469 template <class Integer, typename IntegerSequence, typename Fn, typename ... Args>
00470 void int_for_each(Fn &&f, Args && ... args)
00471 {
00472     detail::int_for_each_helper<Integer, IntegerSequence, Fn, Args...>::apply(std::forward<Fn>(f),
00473                                     std::forward<Args>(args) ...);
00474 }
00475 } // End of namespace util
```

Index

- [~simplicial_complex](#)
 - [casc::simplicial_complex< traits >, 77](#)
- [AbstractSimplicialComplex](#)
 - [casc, 26](#)
- [add_vertex](#)
 - [casc::simplicial_complex< traits >, 77](#)
- [begin](#)
 - [casc::SimplexSet< Complex >, 68](#)
 - [util::range< T >, 55](#)
- [Building the documentation, 13](#)
- [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](#)
 - [getLink, 31](#)
 - [getStar, 31, 32](#)
 - [init_orientation, 32](#)
 - [kneighbors, 33](#)
 - [kneighbors_up, 33, 34](#)
 - [neighbors, 34, 35](#)
 - [neighbors_up, 35, 36](#)
 - [NodeSet, 26](#)
 - [operator!=, 36](#)
 - [operator==, 36](#)
 - [perform_insertion, 37](#)
 - [perform_removal, 37](#)
 - [run_user_callback, 37](#)
 - [set_difference, 39](#)
 - [set_intersection, 39](#)
 - [set_union, 39](#)
 - [to_string, 40](#)
 - [visit_BFS_down, 40](#)
 - [visit_BFS_up, 41](#)
 - [writeDOT, 41](#)
- [CASC License, 5](#)
- [casc::Orientable, 54](#)
- [casc::SimplexMap< Complex >, 62](#)
 - [cLevelIndex, 63](#)
 - [cRevIndex, 63](#)
 - [get, 64](#)
 - [operator<<, 64](#)
 - [RevIndex, 64](#)
- [casc::SimplexSet< Complex >, 65](#)
 - [begin, 68](#)
 - [cbegin, 68](#)
 - [cend, 68](#)
 - [cLevelIndex, 67](#)
 - [cRevIndex, 67](#)
 - [empty, 68](#)
 - [end, 69](#)
 - [erase, 69](#)
 - [find, 70](#)
 - [get, 70, 71](#)
 - [insert, 71](#)
 - [operator<<, 72](#)
 - [RevIndex, 67](#)
 - [SimplexIDLevel, 67](#)
 - [size, 71](#)
- [casc::simplicial_complex< traits >, 72](#)
 - [~simplicial_complex, 77](#)
 - [add_vertex, 77](#)
 - [down, 77, 78](#)
 - [EdgeData, 76](#)
 - [EdgeID, 95](#)
 - [eq, 78, 79](#)
 - [exists, 79](#)
 - [get_cover, 80](#)
 - [get_cover_insert, 80](#)
 - [get_edge_down, 81](#)
 - [get_edge_up, 82](#)
 - [get_level, 83](#)
 - [get_level_id, 83](#)
 - [get_name, 84](#)
 - [get_simplex_down, 85](#)
 - [get_simplex_up, 86, 87](#)
 - [insert, 87, 88](#)
 - [leq, 89](#)
 - [lt, 90](#)
 - [nearBoundary, 90](#)
 - [NodeData, 76](#)
 - [onBoundary, 91](#)
 - [remove, 91, 92](#)
 - [SimplexID, 95](#)
 - [size, 92](#)
 - [up, 94](#)
- [casc::simplicial_complex< traits >::EdgeID< k >, 48](#)
 - [down, 50](#)
 - [EdgeID, 49](#)
 - [up, 50](#)

casc::simplicial_complex< traits >::SimplexID< k >, 57
 cover, 60
 cover_insert, 60
 get_simplex_up, 60, 61
 indices, 61
 operator<<, 62
 SimplexID, 59
 cbegin
 casc::SimplexSet< Complex >, 68
 cend
 casc::SimplexSet< Complex >, 68
 check_orientation
 casc, 26
 clear_orientation
 casc, 27
 cLevelIndex
 casc::SimplexMap< Complex >, 63
 casc::SimplexSet< Complex >, 67
 Colored Abstract Simplicial Complex (CASC) Library, 1
 compute_orientation
 casc, 27
 cover
 casc::simplicial_complex< traits >::SimplexID< k
 >, 60
 cover_insert
 casc::simplicial_complex< traits >::SimplexID< k
 >, 60
 cRevIndex
 casc::SimplexMap< Complex >, 63
 casc::SimplexSet< Complex >, 67
 decimate
 casc, 27
 decimateBackHalf
 casc, 28
 decimateFirstHalf
 casc, 28
 down
 casc::simplicial_complex< traits >, 77, 78
 casc::simplicial_complex< traits >::EdgeID< k >,
 50
 edge_up
 casc, 29
 EdgeData
 casc::simplicial_complex< traits >, 76
 EdgeID
 casc::simplicial_complex< traits >, 95
 casc::simplicial_complex< traits >::EdgeID< k >,
 49
 empty
 casc::SimplexSet< Complex >, 68
 end
 casc::SimplexSet< Complex >, 69
 util::range< T >, 55
 eq
 casc::simplicial_complex< traits >, 78, 79
 erase
 casc::SimplexSet< Complex >, 69
 exists
 casc::simplicial_complex< traits >, 79
 find
 casc::SimplexSet< Complex >, 70
 Frequently Asked Questions, 15
 get
 casc, 29, 30
 casc::SimplexMap< Complex >, 64
 casc::SimplexSet< Complex >, 70, 71
 get_cover
 casc::simplicial_complex< traits >, 80
 get_cover_insert
 casc::simplicial_complex< traits >, 80
 get_edge_down
 casc::simplicial_complex< traits >, 81
 get_edge_up
 casc::simplicial_complex< traits >, 82
 get_level
 casc::simplicial_complex< traits >, 83
 get_level_id
 casc::simplicial_complex< traits >, 83
 get_name
 casc::simplicial_complex< traits >, 84
 get_simplex_down
 casc::simplicial_complex< traits >, 85
 get_simplex_up
 casc::simplicial_complex< traits >, 86, 87
 casc::simplicial_complex< traits >::SimplexID< k
 >, 60, 61
 getClosure
 casc, 30
 getLink
 casc, 31
 getStar
 casc, 31, 32
 include/casc/CASCFunctions.h, 101, 102
 include/casc/CASCTraversals.h, 105, 106
 include/casc/decimate.h, 112, 113
 include/casc/index_tracker.h, 118, 120
 include/casc/Orientable.h, 131, 132
 include/casc/SimplexMap.h, 134, 135
 include/casc/SimplexSet.h, 136, 137
 include/casc/SimplicialComplex.h, 142, 143
 include/casc/stringutil.h, 161, 162
 include/casc/typetraits.h, 162, 163
 include/casc/util.h, 164, 165
 index_tracker, 41
 index_tracker::index_tracker< _T, _d >, 51
 index_tracker::index_tracker< _T, _d >, 50
 index_tracker, 51
 index_tracker::index_tracker_detail, 42
 index_tracker::index_tracker_detail::BTreeNode< _T, _d
 >, 47
 index_tracker::index_tracker_detail::Interval< T >, 52
 operator=, 53
 indices

- casc::simplicial_complex< traits >::SimplexID< k >, 61
- init_orientation
 - casc, 32
- insert
 - casc::SimplexSet< Complex >, 71
 - casc::simplicial_complex< traits >, 87, 88
- int_for_each
 - util, 44
- kneighbors
 - casc, 33
- kneighbors_up
 - casc, 33, 34
- leq
 - casc::simplicial_complex< traits >, 89
- lt
 - casc::simplicial_complex< traits >, 90
- make_range
 - util, 45
- nearBoundary
 - casc::simplicial_complex< traits >, 90
- neighbors
 - casc, 34, 35
- neighbors_up
 - casc, 35, 36
- NodeData
 - casc::simplicial_complex< traits >, 76
- NodeSet
 - casc, 26
- onBoundary
 - casc::simplicial_complex< traits >, 91
- operator!=
 - casc, 36
- operator<<
 - casc::SimplexMap< Complex >, 64
 - casc::SimplexSet< Complex >, 72
 - casc::simplicial_complex< traits >::SimplexID< k >, 62
- operator=
 - index_tracker::index_tracker_detail::Interval< T >, 53
- operator==
 - casc, 36
- perform_insertion
 - casc, 37
- perform_removal
 - casc, 37
- range
 - util::range< T >, 54, 55
- remove
 - casc::simplicial_complex< traits >, 91, 92
- RevIndex
 - casc::SimplexMap< Complex >, 64
- casc::SimplexSet< Complex >, 67
- run_user_callback
 - casc, 37
- set_difference
 - casc, 39
- set_intersection
 - casc, 39
- set_union
 - casc, 39
- SimplexID
 - casc::simplicial_complex< traits >, 95
 - casc::simplicial_complex< traits >::SimplexID< k >, 59
- SimplexIDLevel
 - casc::SimplexSet< Complex >, 67
- size
 - casc::SimplexSet< Complex >, 71
 - casc::simplicial_complex< traits >, 92
- to_string
 - casc, 40
- type_name
 - typetraits.h, 163
- typetraits.h
 - type_name, 163
- up
 - casc::simplicial_complex< traits >, 94
 - casc::simplicial_complex< traits >::EdgeID< k >, 50
- util, 44
 - int_for_each, 44
 - make_range, 45
- util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >, 52
- util::range< T >, 54
 - begin, 55
 - end, 55
 - range, 54, 55
- util::remove_first_val< Integer, InHolder< Integer, I, Is... >, 56
- util::remove_first_val< Integer, IntegerSequence >, 56
- util::reverse_sequence< Integer, IntegerSequence >, 57
- util::type_get< 0, type_holder< Ts... > >, 96
- util::type_get< k, T >, 95
- util::type_get< k, type_holder< Ts... > >, 96
- util::type_holder< T, Ts... >, 97
- util::type_holder< Ts >, 97
- util::type_map< C, V >, 98
- util::type_swap< TUPLE, HOLDER< Ts... > >, 99
- util::type_swap< TUPLE, HOLDER_FULL >, 99
- visit_BFS_down
 - casc, 40
- visit_BFS_up
 - casc, 41
- writeDOT

casc, [41](#)