

CASC

Generated by Doxygen 1.9.4

1 Colored Abstract Simplicial Complex (CASC) Library	1
1.1 Getting Started	1
1.1.1 Prerequisites	1
1.1.2 Installing	2
1.1.3 Documentation	2
1.2 Versioning & Contributing	2
1.3 Authors	2
1.4 License	2
1.5 Acknowledgments	3
2 CASC License	5
2.0.1 GNU LESSER GENERAL PUBLIC LICENSE	5
2.0.2 Preamble	5
2.0.3 TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION	7
2.0.4 END OF TERMS AND CONDITIONS	11
2.0.5 How to Apply These Terms to Your New Libraries	11
3 Building the documentation	13
3.0.1 Documentation for Developers	13
4 Frequently Asked Questions	15
5 Namespace Index	17
5.1 Namespace List	17
6 Data Structure Index	19
6.1 Data Structures	19
7 File Index	21
7.1 File List	21
8 Namespace Documentation	23
8.1 casc Namespace Reference	23
8.1.1 Typedef Documentation	26
8.1.1.1 AbstractSimplicialComplex	26
8.1.2 Function Documentation	26
8.1.2.1 check_orientation()	26
8.1.2.2 clear_orientation()	27
8.1.2.3 compute_orientation()	27
8.1.2.4 decimate()	27
8.1.2.5 decimateBackHalf()	28
8.1.2.6 decimateFirstHalf()	28
8.1.2.7 edge_up()	29
8.1.2.8 get() [1/3]	29
8.1.2.9 get() [2/3]	29

8.1.2.10	get() [3/3]	30
8.1.2.11	getClosure() [1/2]	30
8.1.2.12	getClosure() [2/2]	31
8.1.2.13	getLink() [1/2]	31
8.1.2.14	getLink() [2/2]	32
8.1.2.15	getStar() [1/2]	32
8.1.2.16	getStar() [2/2]	32
8.1.2.17	init_orientation()	33
8.1.2.18	kneighbors() [1/2]	33
8.1.2.19	kneighbors() [2/2]	34
8.1.2.20	kneighbors_up() [1/2]	34
8.1.2.21	kneighbors_up() [2/2]	35
8.1.2.22	neighbors() [1/2]	35
8.1.2.23	neighbors() [2/2]	36
8.1.2.24	neighbors_up() [1/2]	36
8.1.2.25	neighbors_up() [2/2]	37
8.1.2.26	operator!=(())	37
8.1.2.27	operator==(())	38
8.1.2.28	perform_insertion()	38
8.1.2.29	perform_removal()	39
8.1.2.30	run_user_callback()	39
8.1.2.31	set_difference()	39
8.1.2.32	set_intersection()	40
8.1.2.33	set_union()	40
8.1.2.34	to_string()	41
8.1.2.35	visit_BFS_down()	41
8.1.2.36	visit_BFS_up()	42
8.1.2.37	writeDOT()	42
8.2	index_tracker Namespace Reference	42
8.3	index_tracker::index_tracker_detail Namespace Reference	43
8.4	util Namespace Reference	44
8.4.1	Function Documentation	45
8.4.1.1	int_for_each()	45
8.4.1.2	make_range() [1/2]	46
8.4.1.3	make_range() [2/2]	46
9	Data Structure Documentation	49
9.1	index_tracker::index_tracker_detail::BTreeNode< _T, _d > Struct Template Reference	49
9.1.1	Detailed Description	49
9.2	casc::simplicial_complex< traits >::EdgeID< k > Struct Template Reference	50
9.2.1	Detailed Description	51
9.2.2	Constructor & Destructor Documentation	51

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

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

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

9.16.1 Detailed Description	99
9.17 util::type_get< k, type_holder< Ts... > > Struct Template Reference	99
9.17.1 Detailed Description	99
9.18 util::type_holder< Ts > Struct Template Reference	100
9.18.1 Detailed Description	100
9.19 util::type_holder< T, Ts... > Struct Template Reference	100
9.19.1 Detailed Description	101
9.20 util::type_map< C, V > Struct Template Reference	101
9.20.1 Detailed Description	101
9.21 util::type_swap< TUPLE, HOLDER_FULL > Struct Template Reference	102
9.21.1 Detailed Description	102
9.22 util::type_swap< TUPLE, HOLDER< Ts... > > Struct Template Reference	102
9.22.1 Detailed Description	102
10 File Documentation	105
10.1 include/casc/CASCFunctions.h File Reference	105
10.2 CASCFunctions.h	106
10.3 include/casc/CASCTraversals.h File Reference	111
10.4 CASCTraversals.h	113
10.5 include/casc/decimate.h File Reference	122
10.6 decimate.h	123
10.7 include/casc/index_tracker.h File Reference	130
10.8 index_tracker.h	132
10.9 include/casc/Orientable.h File Reference	143
10.10 Orientable.h	144
10.11 include/casc/SimplexMap.h File Reference	147
10.12 SimplexMap.h	148
10.13 include/casc/SimplexSet.h File Reference	150
10.14 SimplexSet.h	151
10.15 include/casc/SimplicialComplex.h File Reference	158
10.16 SimplicialComplex.h	160
10.17 include/casc/stringutil.h File Reference	192
10.18 stringutil.h	192
10.19 include/casc/typetraits.h File Reference	193
10.19.1 Detailed Description	193
10.19.2 Function Documentation	194
10.19.2.1 type_name()	194
10.20 typetraits.h	194
10.21 include/casc/util.h File Reference	195
10.22 util.h	196
Index	203

Chapter 1

Colored Abstract Simplicial Complex (CASC) Library

Master CI: Development CI:

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

1.1 Getting Started

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

1.1.1 Prerequisites

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

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

Doxygen and Graphviz is used to generate the documentation.

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

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

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

1.1.2 Installing

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

```
mkdir build
cd build
```

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

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

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

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

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

1.1.3 Documentation

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

1.2 Versioning & Contributing

We use [Github](#) for versioning. For the versions available, please see the [releases](#). If you find a bug or 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	42
index_tracker::index_tracker_detail	B-tree internal data structures	43
util	Metatemplate programming utilities namespace	44

Chapter 6

Data Structure Index

6.1 Data Structures

Here are the data structures with brief descriptions:

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

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

Chapter 7

File Index

7.1 File List

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

include/casc/CASCFunctions.h	
Contains various functions that operate on simplicial complexes	105
include/casc/CASCTraversals.h	
Implementations of various advanced traversals such as by neighborhood and breadth first search	111
include/casc/decimate.h	
Meta-data aware decimation functions	122
include/casc/index_tracker.h	
B-tree based interval tracker	130
include/casc/Orientable.h	
Data type for orientability	143
include/casc/SimplexMap.h	
SimplexMap data structure and associated convenience functions	147
include/casc/SimplexSet.h	
SimplexSet data structure and associated convenience functions	150
include/casc/SimplicialComplex.h	
This header contains the main CASC data structure and associated components	158
include/casc/stringutil.h	
String utilities for CASC	192
include/casc/typetraits.h	
Helper functions for debugging template types	193
include/casc/util.h	
Metatemplate pack expansion helpers	195

Chapter 8

Namespace Documentation

8.1 casc Namespace Reference

Namespace for everything CASC.

Data Structures

- struct [Orientable](#)
Class representing the orientation.
- struct [SimplexMap](#)
A multimap to represent a map of simplex indices to a set of simplices.
- struct [SimplexSet](#)
A multiset to store simplices in a [simplicial_complex](#).
- class [simplicial_complex](#)
The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring.

Typedefs

- template<typename KeyType , typename ... Ts>
using [AbstractSimplicialComplex](#) = [simplicial_complex](#)< detail::simplicial_complex_traits_default< KeyType, Ts... > >
- template<typename T >
using **NodeSet** = std::unordered_set< T, simplex_set_detail::hashSimplexID< T > >
Helpful alias defining a unordered_set of simplices. See also hashSimplexID.

Functions

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

- `template<typename Complex >`
`void getClosure (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)`
Gets the closure of a simplex set.
- `template<typename Complex , typename Simplex >`
`void getClosure (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)`
Compute the closure of a simplex.
- `template<typename Complex >`
`void getLink (Complex &F, casc::SimplexSet< Complex > &S, casc::SimplexSet< Complex > &dest)`
Gets the link of a [SimplexSet](#).
- `template<typename Complex , typename Simplex >`
`void getLink (Complex &F, Simplex &s, casc::SimplexSet< Complex > &dest)`
Gets the link of a simplex.
- `template<typename Complex >`
`void writeDOT (const std::string &filename, Complex &F)`
Writes out the topology of an ASC into the dot format.
- `template<typename Visitor , typename SimplexID >`
`void visit_BFS_up (Visitor &&v, typename SimplexID::complex &F, SimplexID s)`
Traverse BFS up the complex and apply a visitor function to each simplex visited.
- `template<typename Visitor , typename SimplexID >`
`void visit_BFS_down (Visitor &&v, typename SimplexID::complex &F, SimplexID s)`
Traverse BFS down the complex and apply a visitor function to each simplex visited.
- `template<typename Visitor , typename EdgeID >`
`void edge_up (Visitor &&v, typename EdgeID::complex &F, EdgeID s)`
Traverse across edges BFS.
- `template<class Complex , std::size_t level, class InsertIter >`
`void neighbors (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)`
Push the immediate face neighbors into the provided iterator.
- `template<class Complex , class SimplexID , class InsertIter >`
`void neighbors (Complex &F, SimplexID nid, InsertIter iter)`
This is a helper function to assist neighbors to automatically deduce the integral level.
- `template<class Complex , std::size_t level, class InsertIter >`
`void neighbors_up (Complex &F, typename Complex::template SimplexID< level > nid, InsertIter iter)`
Push the immediate coface neighbors into the provided iterator.
- `template<class Complex , class SimplexID , class InsertIter >`
`void neighbors_up (Complex &F, SimplexID nid, InsertIter iter)`
This is a helper function to assist neighbors to automatically deduce the integral level.
- `template<class Complex , std::size_t level, typename Iterator >`
`void kneighbors_up (Complex &F, int ring, std::set< typename Complex::template SimplexID< level > > &nbrs, 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 > &nbrs)`
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 > > &nbrs, 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 > &nbrs)`
Helper function to help [kneighbors](#) to deduce the integral level of SimplexID.
- `template<typename Complex >`
`void perform_removal (Complex &F, casc::SimplexSet< Complex > &S)`
Remove simplex in [SimplexSet](#) S from complex F.

- template<typename Complex >
void [perform_insertion](#) (Complex &F, typename decimation_detail::SimplexDataSet< Complex >::type &S)
Insert all simplices in [SimplexSet](#) S into complex F
- template<typename Complex , template< typename > class Callback>
void [run_user_callback](#) (Complex &F, [casc::SimplexMap](#)< Complex > &S, Callback< Complex > &&clbk, typename decimation_detail::SimplexDataSet< Complex >::type &rv)
Run the user specified callback function.
- template<typename Complex , typename Simplex , template< typename > class Callback>
void [decimate](#) (Complex &F, Simplex s, Callback< Complex > &&clbk)
Decimate a simplex of any dimension while considering any meta-data stores on decimated simplices.
- template<typename Complex , typename Simplex >
Complex::KeyType [decimateFirstHalf](#) (Complex &F, Simplex s, [SimplexMap](#)< Complex > &simplexMap)
Given a simplex to decimate generate a pre-post mapping.
- template<typename Complex >
void [decimateBackHalf](#) (Complex &F, [SimplexMap](#)< Complex > &simplexMap, typename decimation_detail::SimplexDataSet< Complex >::type &rv)
Given a simplexMap and mapped resulting data execute the decimation.
- template<typename Complex >
void [init_orientation](#) (Complex &F)
Initialize the partial ordering of the simplex edges.
- template<typename Complex >
void [clear_orientation](#) (Complex &F)
Clear the orientation of the facets.
- template<typename Complex >
std::tuple< int, bool, bool > [compute_orientation](#) (Complex &F)
Initializes and calculates the orientation of a [simplicial_complex](#).
- template<typename Complex >
std::tuple< int, bool, bool > [check_orientation](#) (Complex &F)
Checks for self consistent orientation and fill in missing orientations.
- template<std::size_t k, typename Complex >
static auto & [get](#) ([SimplexMap](#)< Complex > &S)
Get the map for a simplex dimension.
- template<std::size_t k, typename Complex >
static auto & [get](#) (const [SimplexMap](#)< Complex > &S)
This is an overloaded member function, provided for convenience. It differs from the above function only in what argument(s) it accepts.
- template<std::size_t k, typename Complex >
static auto & [get](#) ([SimplexSet](#)< Complex > &S)
Get the NodeSet for a simplex dimension from a [SimplexSet](#).
- template<std::size_t k, typename Complex >
static auto & [get](#) (const [SimplexSet](#)< Complex > &S)
- template<typename Complex >
bool [operator==](#) (const [SimplexSet](#)< Complex > &lhs, const [SimplexSet](#)< Complex > &rhs)
Compare if the sets are equivalent.
- template<typename Complex >
bool [operator!=](#) (const [SimplexSet](#)< Complex > &lhs, const [SimplexSet](#)< Complex > &rhs)
Compare if the sets are not equivalent.
- template<typename Complex >
static void [set_union](#) (const [SimplexSet](#)< Complex > &A, const [SimplexSet](#)< Complex > &B, [SimplexSet](#)< Complex > &dest)
Compute the set union.
- template<typename Complex >
static void [set_intersection](#) (const [SimplexSet](#)< Complex > &A, const [SimplexSet](#)< Complex > &B, [SimplexSet](#)< Complex > &dest)

Compute the set intersection.

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

Compute the set difference.

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

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

8.1.1 Typedef Documentation

8.1.1.1 AbstractSimplicialComplex

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

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

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

8.1.2 Function Documentation

8.1.2.1 check_orientation()

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

Parameters

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

Template Parameters

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

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

S	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 nieghbors 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 `kneighbors_up()` [1/2]

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

Parameters

in	<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 kneighbors_up() [2/2]

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

Parameters

in	<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
----------------	--------------------------------

Template Parameters

<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

\leftarrow	{ description }
\overleftarrow{T}	
\leftarrow	{ description }
\overleftarrow{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 <code>k-1</code> to a simplex of size <code>k</code> .
----------------	--

9.2.2 Constructor & Destructor Documentation

9.2.2.1 `EdgeID()` [1/2]

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

Parameters

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

9.2.2.2 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

\leftrightarrow — T	Typename of the indices
\leftrightarrow — 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

in	rhs	The right hand side
----	-----	---------------------

Returns

Reference to this

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

- include/casc/[index_tracker.h](#)

9.6 casc::Orientable Struct Reference

Class representing the orientation.

```
#include <Orientable.h>
```

Data Fields

- int **orientation**
Integer representing +/- 1 orientation.

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

- include/casc/[Orientable.h](#)

9.7 util::range< T > Struct Template Reference

A range object to support range based for loops.

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

Public Member Functions

- template<class C >
[range](#) (C &&c)
Construct a range for a container class.
- [range](#) (T b, T e)
Construct a range from an iterator.
- T [begin](#) ()
Get the beginning iterator.
- T [end](#) ()
Get the end iterator.

9.7.1 Detailed Description

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

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

Template Parameters

<i>T</i>	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 <code>begin()</code> and <code>end()</code> .
----	----------	--

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

in	<code>p</code>	The <code>NodePtr</code> 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	<code>rhs</code>	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	s	Array of keys to follow
----	---	-------------------------

Template Parameters

j	Number of keys
---	----------------

Returns

The simplex up

9.11.3.5 `get_simplex_up()` [3/3]

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

Parameters

in	arr	Array of keys to follow
----	-----	-------------------------

Template Parameters

j	Number of keys
---	----------------

Returns

The simplex up

9.11.3.6 `indices()`

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

Parameters

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

Returns

Array containing the name of 'id'.

9.11.4 Friends And Related Function Documentation

9.11.4.1 `operator<<`

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

Parameters

	<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** = typename [util::remove_first_val](#)< std::size_t, [LevelIndex](#) >::type
Index sequence starting at 1.
- using **RevIndex** = typename [util::reverse_sequence](#)< std::size_t, [LevelIndex](#) >::type
Reversed Index sequence.
- using **cRevIndex** = typename [util::reverse_sequence](#)< std::size_t, [cLevelIndex](#) >::type
Reversed index sequence stops at 1.
- using **type_this** = [SimplexMap](#)< Complex >
Typename of this object.

Public Member Functions

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

Friends

- `std::ostream & operator<< (std::ostream &output, const SimplexMap< Complex > &S)`
Print the [SimplexMap](#).

9.12.1 Detailed Description

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

Template Parameters

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

9.12.2 Member Function Documentation

9.12.2.1 `get()` [1/2]

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

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

Template Parameters

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

Returns

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

9.12.2.2 `get()` [2/2]

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

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

9.12.3 Friends And Related Function Documentation

9.12.3.1 `operator<<`

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

Parameters

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

Public Member Functions

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

Get the simplex of interest.

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

Get the simplex of interest.

- `template<std::size_t k>`
`auto end ()`

Get the past-the-end iterator.

- `template<std::size_t k>`
`auto cend () const`

Get the past-the-end iterator.

- `template<std::size_t k>`
`auto begin ()`

Get an 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 ()`
- `template<std::size_t k>`
`auto & get () const`

Data Fields

- `util::type_map< SimplexIDLevel, NodeSet >::type tupleSet`
Tuple of NodeSets per level.

Friends

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

9.13.1 Detailed Description

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

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

Template Parameters

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

9.13.2 Member Function Documentation

9.13.2.1 begin()

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

Template Parameters

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

Returns

Returns an iterator to the first element.

9.13.2.2 cbegin()

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

Template Parameters

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

Returns

Returns an iterator to the first element.

9.13.2.3 cend()

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

Template Parameters

<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.

9.13.2.4 `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.

9.13.2.5 `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.

9.13.2.6 `erase()` [1/2]

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

Parameters

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

9.13.2.7 `erase()` [2/2]

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

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

Parameters

in	s	Simplex to remove.
----	---	--------------------

Template Parameters

k	Simplex dimension of 's'.
---	---------------------------

9.13.2.8 find() [1/2]

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

Parameters

in	s	The simplex to search for.
----	---	----------------------------

Template Parameters

k	Simplex dimension of 's'.
---	---------------------------

Returns

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

9.13.2.9 find() [2/2]

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

Parameters

in	s	The simplex to search for.
----	---	----------------------------

Template Parameters

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

Returns

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

9.13.2.10 `insert()` [1/2]

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

Parameters

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

9.13.2.11 `insert()` [2/2]

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

Parameters

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

Template Parameters

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

9.13.2.12 `size()`

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

Template Parameters

k	Simplex dimension to query
-----	----------------------------

Returns

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

9.13.3 Friends And Related Function Documentation

9.13.3.1 `operator<<`

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

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

Parameters

	<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 Inserter >`
`void get_cover_insert (const SimplexID< k > id, Inserter pos) const`
Insert the coboundary keys of a simple into an inserter.
- `template<std::size_t k, class Lambda >`
`void get_cover (const SimplexID< k > id, Lambda fn) const`
Apply a lambda function to the coboundary keys.
- `template<std::size_t k>`
`std::vector< KeyType > get_cover (const SimplexID< k > id) const`
Get the coboundary keys of a simplex.
- `template<std::size_t k>`
`std::set< SimplexID< k+1 > > up (const std::set< SimplexID< k > > &&simplices) const`
Get the coboundary of a set of simplices.
- `template<std::size_t k>`
`std::set< SimplexID< k+1 > > up (const std::set< SimplexID< k > > &simplices) const`
Get the coboundary of a set of simplices.
- `template<std::size_t k>`
`std::set< SimplexID< k+1 > > up (const SimplexID< k > nid) const`
Get the coboundary of a simplex.
- `template<std::size_t k, class InsertIter >`
`void up (const std::set< SimplexID< k > > &&simplices, InsertIter iter) const`
- `template<std::size_t k, class InsertIter >`
`void up (const std::set< SimplexID< k > > &simplices, InsertIter iter) const`
- `template<std::size_t k, class InsertIter >`
`void up (const SimplexID< k > simplex, InsertIter iter) const`
- `template<std::size_t k>`
`std::set< SimplexID< k-1 > > down (const std::set< SimplexID< k > > &&simplices) const`
Get the boundary of a set of simplices.
- `template<std::size_t k>`
`std::set< SimplexID< k-1 > > down (const std::set< SimplexID< k > > &simplices) const`
Get the boundary of a set of simplices.
- `template<std::size_t k>`
`std::set< SimplexID< k-1 > > down (const SimplexID< k > simplex) const`
Get the boundary of a simplex.

- `template<std::size_t k, class InsertIter >`
`void down (const std::set< SimplexID< k > > &&simplices, InsertIter iter) const`
- `template<std::size_t k, class InsertIter >`
`void down (const std::set< SimplexID< k > > &simplices, InsertIter iter) const`
- `template<std::size_t k, class InsertIter >`
`void down (const SimplexID< k > simplex, InsertIter iter) const`
- `template<std::size_t k>`
`EdgeID< k+1 > get_edge_up (SimplexID< k > simplex, KeyType a)`
Gets the edge up from a simplex.
- `template<std::size_t k>`
`EdgeID< k > get_edge_down (SimplexID< k > simplex, KeyType a)`
Gets the edge down from a simplex.
- `template<std::size_t k>`
`EdgeID< k+1 > get_edge_up (SimplexID< k > simplex, KeyType a) const`
Gets the edge up from a simplex.
- `template<std::size_t k>`
`EdgeID< k > get_edge_down (SimplexID< k > simplex, KeyType a) const`
Gets the edge down from a simplex.
- `template<std::size_t k>`
`bool exists (const KeyType(&s)[k]) const`
Check whether a simplex with some name exists.
- `template<std::size_t k>`
`std::size_t size () const`
Get the number of simplices of dimension 'k'.
- `template<std::size_t k>`
`auto get_level_id ()`
Create an iterator to traverse the SimplexIDs of a dimension.
- `template<std::size_t k>`
`auto get_level_id () const`
Create an iterator to traverse the SimplexIDs of a dimension.
- `template<std::size_t k>`
`auto get_level ()`
Create an iterator to traverse the simplex data of a dimension.
- `template<std::size_t k>`
`auto get_level () const`
Create an iterator to traverse the simplex data of a dimension.
- `template<std::size_t k>`
`std::size_t remove (const KeyType(&s)[k])`
Remove a simplex and all dependent simplices by name.
- `template<std::size_t k>`
`std::size_t remove (const std::array< KeyType, k > &s)`
Remove a simplex and all dependent simplices by name.
- `template<std::size_t k>`
`std::size_t remove (SimplexID< k > s)`
Remove a simplex and all dependent simplices by SimplexID.
- `template<std::size_t k>`
`bool onBoundary (const SimplexID< k > s) const`
Checks whether a simplex is on a boundary.
- `template<std::size_t level>`
`bool nearBoundary (const SimplexID< level > s) const`
Checks whether a simplex is near a boundary.
- `template<std::size_t L, std::size_t R>`
`bool leq (SimplexID< L > lhs, SimplexID< R > rhs) const`

Less than or equal to comparison operator of two SimplexIDs.

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

Equality comparison of two simplices.

- `template<std::size_t k>`
`bool eq (SimplexID< k > lhs, SimplexID< k > rhs) const`

Equality comparison of two simplices.

- `template<std::size_t L, std::size_t R>`
`bool lt (SimplexID< L > lhs, SimplexID< R > rhs) const`

Less than comparison of simplices.

Static Public Attributes

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

Total number of levels in the complex.

- static constexpr `std::size_t topLevel` = `numLevels-1`

Dimension of the simplicial complex.

- static constexpr `std::size_t bdryLevel` = `numLevels-2`

Dimension of boundaries.

Friends

- struct `SimplexID`
- struct `EdgeID`

9.14.1 Detailed Description

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

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

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

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

Template Parameters

<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 simplices.
----------	---------------------------------

Returns

The set of boundary simplices.

9.14.4.5 `down()` [3/3]

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

Parameters

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

Template Parameters

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

Returns

The set of boundary simplices.

9.14.4.6 `eq()` [1/2]

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

Parameters

in	<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>Insertter</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

<i>k</i>	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

<i>k</i>	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

<i>k</i>	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

<i>k</i>	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

<code>k</code>	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

<code>in</code>	<code>id</code>	<code>SimplexID</code> 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	s	Name of the simplex to find.
----	---	------------------------------

Template Parameters

n	Dimension of simplex s.
---	-------------------------

Returns

SimplexID of node corresponding to 's'.

9.14.4.28 get_simplex_up() [3/4]

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

Parameters

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

Template Parameters

i	The size of simplex 'id'.
---	---------------------------

Returns

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

9.14.4.29 `get_simplex_up()` [4/4]

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

Parameters

in	<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

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

Template Parameters

<i>n</i>	Dimension of simplex 's'.
----------	---------------------------

9.14.4.34 `leq()`

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

Parameters

in	<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

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

Template Parameters

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

Returns

True if the simplex or any subsimplices are onBoundary.

9.14.4.37 onBoundary()

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

Parameters

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

Template Parameters

<i>k</i>	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

<code>in</code>	<code>s</code>	C-style array with the name of the simplex to remove.
-----------------	----------------	---

Template Parameters

<code>k</code>	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

<code>in</code>	<code>s</code>	<code>std::array</code> with the name of the simplex to remove.
-----------------	----------------	---

Template Parameters

<code>k</code>	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

<i>in</i>	<i>s</i>	SimplexID of the simplex to remove.
-----------	----------	---

Template Parameters

<i>k</i>	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 Function Documentation**9.14.5.1 EdgeID**

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

EdgeID is a friend to [simplicial_complex](#)

9.14.5.2 SimplexID

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

SimplexID is a friend of [simplicial_complex](#)

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

- include/casc/[SimplicialComplex.h](#)

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

Helper to get the kth element from a [type_holder](#).

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

9.15.1 Detailed Description

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

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

Template Parameters

<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 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 $\mathbb{V}<T>$ 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.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file CASCFunctions.h
27  * @brief Contains various functions that operate on simplicial complexes
28  */
29
30 #pragma once
31
32 #include <iostream>
33 #include <fstream>
34 #include "SimplicialComplex.h"
35 #include "CASCTraversals.h"
36 #include "SimplexSet.h"
37 #include "stringutil.h"
38
39 namespace casc
40 {
41     /// @cond detail
42     /// Namespace for templated helpers of convenience functions
43     namespace func_detail
44     {
45
46     /**
47      * @brief Visitor for grabbing simplices in a BFS traversal. See also
48      * visit_BFS_up and visit_BFS_down.
49      *
50      * @tparam Complex Typename of the complex.
51      */
52     template <typename Complex>
53     struct SimplexAggregator
54     {
55         /// Alias for a typed SimplexSet
56         using SimplexSet = typename casc::SimplexSet<Complex>;
57
58         /**
59          * @brief Constructor of the aggregator.
60          *
61          * @param p Pointer to SimplexSet to fill.
62          */
63         SimplexAggregator(SimplexSet* p) : pLevels(p) {}
64
65         /**
66          * @brief Overloaded visit function to append different dimensioned
67          * SimplexIDs into the SimplexSet.
68          *
69          * @param F The complex of interest
70          * @param[in] s SimplexID to visit.
71          *
72          * @tparam k Dimension of the simplex.
73          *
74          * @return True if the traversal should continue.
75          */
76         template <std::size_t k>
77         bool visit(Complex &, typename Complex::template SimplexID<k> s)
78         {
79             // If the simplex isn't there, insert it.
80             if (pLevels->find(s) == pLevels->template end<k>())
81             {
82                 pLevels->insert(s);
83             }
84         }
85     };
86
87     }
88     /// @endcond
89 }

```

```

83         return true;
84     }
85     else
86     {
87         // Everything after has been found already
88         return false;
89     }
90 }
91 private:
92     SimplexSet* pLevels;
93 };
94
95 /**
96  * @brief      Helper for computing the star of a set of simplices.
97  *
98  * @tparam      Complex  Typename of the simplicial complex.
99  */
100 template <typename Complex>
101 struct StarHelper
102 {
103     /**
104      * @brief      Iterate over the SimplexSet and compute the star.
105      *
106      * @param      F        Complex of interest.
107      * @param      S        SimplexSet of simplices to compute the star of.
108      * @param      dest     SimplexSet where the star should go.
109      *
110      * @tparam      k        Dimension of the current simplex dimension to traverse.
111      */
112     template <std::size_t k>
113     static void apply(Complex &F,
114                      casc::SimplexSet<Complex> &S,
115                      casc::SimplexSet<Complex> &dest)
116     {
117         auto s = casc::get<k>(S);
118         for (auto simplex : s)
119         {
120             visit_BFS_up(SimplexAggregator<Complex>(&dest), F, simplex);
121         }
122     }
123 };
124
125 /**
126  * @brief      Helper for computing the closure of a set of simplices.
127  *
128  * @tparam      Complex  Typename of the simplicial complex.
129  */
130 template <typename Complex>
131 struct ClosureHelper
132 {
133     /**
134      * @brief      Iterate over the SimplexSet and compute the closure
135      *
136      * @param      F        Complex of interest.
137      * @param      S        SimplexSet of simplices to compute the closure of.
138      * @param      dest     SimplexSet where the closure should go.
139      *
140      * @tparam      k        Dimension of the current simplex dimension to traverse.
141      */
142     template <std::size_t k>
143     static void apply(Complex &F,
144                      casc::SimplexSet<Complex> &S,
145                      casc::SimplexSet<Complex> &dest)
146     {
147         auto s = casc::get<k>(S);
148         for (auto simplex : s)
149         {
150             visit_BFS_down(SimplexAggregator<Complex>(&dest), F, simplex);
151         }
152     }
153 };
154
155 /**
156  * @brief      Visitor for printing connectivity of the simplicial complex.
157  *
158  * @tparam      Complex  Typename of the simplicial complex.
159  */
160 template <typename Complex>
161 struct GraphVisitor
162 {
163     /// ostream to write out to.
164     std::ostream &fout;
165
166     /**
167      * @brief      Constructor
168      *
169      * @param      os      Ostream to print to.
170      */

```

```

170     */
171     GraphVisitor(std::ostream &os) : fout(os) {}
172
173     /**
174     * @brief      Generic visitor prints the simplices and edge connectivity.
175     *
176     * @param[in]  F          Complex of interest.
177     * @param[in]  s          Simplex to visit.
178     *
179     * @tparam     level      Dimension of the simplex.
180     *
181     * @return     True
182     */
183     template <std::size_t level>
184     bool visit(const Complex &F, typename Complex::template SimplexID<level> s)
185     {
186         auto name = to_string(F.get_name(s));
187
188         auto covers = F.get_cover(s);
189         for (auto cover : covers)
190         {
191             auto edge = F.get_edge_up(s, cover);
192             auto nextName = to_string(F.get_name(edge.up()));
193             if ((*edge).orientation == 1)
194             {
195                 fout << "    \"\" < name < \"\\\" -> \"\"
196                     << nextName < \"\"\" < std::endl;
197             }
198             else
199             {
200                 fout << "    \"\" < nextName < \"\\\" -> \"\"
201                     << name < \"\"\" < std::endl;
202             }
203         }
204         return true;
205     }
206
207     /**
208     * @brief      Explicit specialization for visiting the second to top level
209     *              simplices.
210     *
211     * @param[in]  F          Complex of interest
212     * @param[in]  s          Simplex to visit.
213     *
214     * @return     True;
215     */
216     bool visit(const Complex &F, typename Complex::template SimplexID<Complex::topLevel-1> s)
217     {
218
219         auto name = to_string(F.get_name(s));
220         auto covers = F.get_cover(s);
221         for (auto cover : covers)
222         {
223             auto edge = F.get_edge_up(s, cover);
224             auto nextName = to_string(F.get_name(edge.up()));
225             auto orient = (*edge.up()).orientation;
226             if (orient == 1)
227             {
228                 nextName = "+" + nextName;
229             }
230             else
231             {
232                 nextName = "-" + nextName;
233             }
234             if ((*edge).orientation == 1)
235             {
236                 fout << "    \"\" < name < \"\\\" -> \"\"
237                     << nextName < \"\"\" < std::endl;
238             }
239             else
240             {
241                 fout << "    \"\" < nextName < \"\\\" -> \"\"
242                     << name < \"\"\" < std::endl;
243             }
244         }
245         return true;
246     }
247
248     /**
249     * @brief      Explicit specialization for visiting the facets of the
250     *              *complex.
251     *
252     * @param[in]  F          Complex of interest.
253     * @param[in]  s          Simplex to visit.
254     */
255     void visit(const Complex &, typename Complex::template SimplexID<Complex::topLevel>) {}
256 };

```

```

257
258 /**
259  * @brief      Generic template for printing out DOT meta info.
260  *
261  * @tparam      Complex  Typename of the complex.
262  * @tparam      K        Dimension to go through.
263  */
264 template <typename Complex, typename K>
265 struct DotHelper {};
266
267 /**
268  * @brief      Partial specialization for listing names of simplices.
269  *
270  * @tparam      Complex  Typename of the complex.
271  * @tparam      k        Simplex dimension to traverse.
272  */
273 template <typename Complex, std::size_t k>
274 struct DotHelper<Complex, std::integral_constant<std::size_t, k> >
275 {
276     /**
277     * @brief      Print out a list of simplices in a simplex dimension.
278     *
279     * @param      fout    Stream to print to.
280     * @param[in]   F       Complex of interest.
281     */
282     static void printlevel(std::ofstream &fout, const Complex &F)
283     {
284         auto nodes = F.template get_level_id<k>();
285         fout << "subgraph cluster_" << k << " {\n"
286             << "label=\"Level " << k << "\"\n";
287         for (auto node : nodes)
288         {
289             fout << "\"\" << to_string(F.get_name(node)) << "\"";
290         }
291         fout << "\n}\n";
292         DotHelper<Complex, std::integral_constant<std::size_t, k+1> >::printlevel(fout, F);
293     }
294 };
295
296 /**
297  * @brief      List the names of simplices at the top level
298  *
299  * @tparam      Complex  Typename of the complex.
300  */
301 template <typename Complex>
302 struct DotHelper<Complex, std::integral_constant<std::size_t, Complex::topLevel> >
303 {
304     /**
305     * @brief      Print out a list of facets of the complex.
306     *
307     * @param      fout    Stream to print to.
308     * @param[in]   F       Complex of interest.
309     */
310     static void printlevel(std::ofstream &fout, const Complex &F)
311     {
312         auto nodes = F.template get_level_id<Complex::topLevel>();
313         fout << "subgraph cluster_" << Complex::topLevel << " {\n"
314             << "label=\"Level " << Complex::topLevel << "\"\n";
315         for (auto node : nodes)
316         {
317             auto orient = (*node).orientation;
318             if (orient == 1)
319             {
320                 fout << "\"+ ";
321             }
322             else
323             {
324                 fout << "\"- ";
325             }
326             fout << to_string(F.get_name(node)) << "\"";
327         }
328         fout << "\n}\n";
329     }
330 };
331 } // end namespace func_detail
332 /// @endcond
333
334 /**
335  * @brief      Gets the star of a SimplexSet.
336  *
337  * @param[in]   F        Complex of interest.
338  * @param[in]   S        SimplexSet to compute the star of.
339  * @param[out]  dest     Destination SimplexSet.
340  *
341  * @tparam      Complex  Typename of the complex.
342  */
343 template <typename Complex>

```

```

344 void getStar(Complex &F, casc::SimplexSet<Complex> &S,
345             casc::SimplexSet<Complex> &dest)
346 {
347     using SimplexSet = typename casc::SimplexSet<Complex>;
348     using RevIndex   = typename SimplexSet::cRevIndex;
349
350     // Start at the top and work up. We can assume that if we've seen it then
351     // everything after has been added.
352     util::int_for_each<std::size_t, RevIndex>(
353         func_detail::StarHelper<Complex>(), F, S, dest);
354 }
355
356 /**
357  * @brief      Gets the star of a simplex.
358  *
359  * @param[in]  F      Complex of interest.
360  * @param      s      Simplex to get the star of.
361  * @param[out] dest   Destination SimplexSet.
362  *
363  * @tparam     Complex Typename of the complex.
364  * @tparam     Simplex  Typename of the simplex.
365  */
366 template <typename Complex, typename Simplex>
367 void getStar(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
368 {
369     visit_BFS_up(func_detail::SimplexAggregator<Complex>(&dest), F, s);
370 }
371
372 /**
373  * @brief      Gets the closure of a simplex set.
374  *
375  * @param[in]  F      Complex of interest.
376  * @param[in]  S      SimplexSet to compute the closure of.
377  * @param[out] dest   Destination SimplexSet
378  *
379  * @tparam     Complex Typename of the complex.
380  */
381 template <typename Complex>
382 void getClosure(Complex &F, casc::SimplexSet<Complex> &S,
383               casc::SimplexSet<Complex> &dest)
384 {
385     using SimplexSet = typename casc::SimplexSet<Complex>;
386     using LevelIndex = typename SimplexSet::cLevelIndex;
387     // Start at the bottom and work down.
388     // We can assume that everything below has been looked at.
389     util::int_for_each<std::size_t, LevelIndex>(
390         func_detail::ClosureHelper<Complex>(), F, S, dest);
391 }
392
393 /**
394  * @brief      Compute the closure of a simplex.
395  *
396  * @param[in]  F      Complex of interest.
397  * @param[in]  s      Simplex of interest.
398  * @param[out] dest   Destination SimplexSet.
399  *
400  * @tparam     Complex Typename of the complex.
401  * @tparam     Simplex  Typename of the simplex.
402  */
403 template <typename Complex, typename Simplex>
404 void getClosure(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
405 {
406     visit_BFS_down(func_detail::SimplexAggregator<Complex>(&dest), F, s);
407 }
408
409 /**
410  * @brief      Gets the link of a SimplexSet.
411  *
412  * @param[in]  F      Complex of interest.
413  * @param[in]  S      SimplexSet to get the link of.
414  * @param[out] dest   Destination SimplexSet.
415  *
416  * @tparam     Complex Typename of the complex.
417  */
418 template <typename Complex>
419 void getLink(Complex &F, casc::SimplexSet<Complex> &S,
420            casc::SimplexSet<Complex> &dest)
421 {
422     using SimplexSet = typename casc::SimplexSet<Complex>;
423
424     SimplexSet star;
425     SimplexSet closure;
426     SimplexSet closeStar;
427     SimplexSet starClose;
428     getStar(F, S, star);
429     getClosure(F, star, closeStar);
430 }

```

```

431     getClosure(F, S, closure);
432     getStar(F, closure, starClose);
433     casc::set_difference(closeStar, starClose, dest);
434 }
435
436 /**
437  * @brief      Gets the link of a simplex
438  *
439  * @param      F      Complex of interest.
440  * @param      s      Simplex of interest.
441  * @param      dest    Destination SimplexSet.
442  *
443  * @tparam     Complex Typename of the complex.
444  * @tparam     Simplex Typename of the simplex.
445  */
446 template <typename Complex, typename Simplex>
447 void getLink(Complex &F, Simplex &s, casc::SimplexSet<Complex> &dest)
448 {
449     using SimplexSet = typename casc::SimplexSet<Complex>;
450     SimplexSet star;
451     SimplexSet closure;
452     SimplexSet closeStar;
453     SimplexSet starClose;
454     getStar(F, s, star);
455     getClosure(F, star, closeStar);
456
457     getClosure(F, s, closure);
458     getStar(F, closure, starClose);
459     casc::set_difference(closeStar, starClose, dest);
460 }
461
462 /**
463  * @brief      Writes out the topology of an ASC into the dot format.
464  *
465  * The resulting dot file can be rendered into an image using tools such as
466  * GraphViz.
467  * ~~~~~{.sh}
468  * dot -Tpng input.dot > output.png
469  * ~~~~~
470  *
471  * @param[in]  filename  Filename to write out to.
472  * @param[in]  F          Simplicial complex to generate the DOT of.
473  *
474  * @tparam     Complex   Typename of the simplicial complex.
475  */
476 template <typename Complex>
477 void writeDOT(const std::string &filename, Complex &F)
478 {
479     // TODO: Put back the const F (0)
480     std::ofstream fout(filename);
481     if (!fout.is_open())
482     {
483         std::cerr << "File '" << filename
484                   << "' could not be written to." << std::endl;
485         fout.close();
486         exit(1);
487     }
488
489     fout << "digraph {\n"
490          << "    node [shape = record,height = .1]\n"
491          << "    splines=line;\n"
492          << "    dpi=300;\n";
493     auto v = func_detail::GraphVisitor<Complex>(fout);
494     visit_BFS_up(v, F, F.get_simplex_up());
495
496     // List the simplices
497     func_detail::DotHelper<Complex,
498                          std::integral_constant<std::size_t, 0> >::printlevel(fout, F);
499     fout << "}\n";
500     fout.close();
501 }
502 } // 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.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file CASCTraversals.h
27  * @brief Implementations of various advanced traversals such as by neighborhood
28  * and breadth first search.
29  */
30
31 #pragma once
32
33 #include <set>
34 #include <vector>
35 #include <iostream>
36 #include <string>
37 #include <type_traits>
38 #include <utility>
39 #include <casc/casc>
40
41 namespace casc
42 {
43     /// @cond detail
44     /// Visitor design pattern helper templates
45     namespace visitor_detail
46     {
47
48         /**
49          * @brief General template for BFS up helper.
50          *
51          * @tparam Visitor Type of visitor functor.
52          * @tparam Traits Traits of the BFS traversal.
53          * @tparam Complex Typename of the simplicial_complex.
54          * @tparam K Current simplex dimension to traverse.
55          */
56         template <typename Visitor, typename Traits, typename Complex, typename K>
57         struct BFS_Up_Node {};
58
59         /**
60          * @brief Partial specialization for BFS up helper for non facet
61          * dimensions.
62          *
63          * @tparam Visitor Type of visitor functor.
64          * @tparam Traits Traits of the BFS traversal.
65          * @tparam Complex Typename of the simplicial_complex.
66          * @tparam k Current simplex dimension to traverse.
67          */
68         template <typename Visitor, typename Traits, typename Complex, std::size_t k>
69         struct BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
70         {
71             /// Simplex dimension currently traversed
72             static constexpr auto level = k;
73             /// Typename of the current simplex
74             using CurrSimplexID = typename Complex::template SimplexID<level>;
75             /// Typename of coboundary simplices
76             using NextSimplexID = typename Complex::template SimplexID<level+1>;
77             /// Container to use to hold coboundary simplices for next recursion.
78             template <typename T> using Container = typename Traits::template Container<T>;
79
80             /// Alias for the recursive call
81             using BFS_Up_Node_Next = BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t,
82             level+1> >;

```

```

82
83 /**
84  * @brief      Visit simplices in the current dimension and continue.
85  *
86  * @param[in]  v          Visitor functor.
87  * @param[in]  F          The simplicial_complex to traverse.
88  * @param[in]  begin      Iterator to simplices to traverse.
89  * @param[in]  end        Iterator to end of simplices to traverse.
90  *
91  * @tparam     Iterator   Typename of the iterator.
92  */
93 template <typename Iterator>
94 static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
95 {
96     Container<NextSimplexID> next;
97
98     for (auto curr = begin; curr != end; ++curr)
99     {
100         if (v.visit(F, *curr))
101         {
102             F.get_cover(*curr, [&](typename Complex::KeyType a)
103             {
104                 auto id = F.get_simplex_up(*curr, a);
105                 next.insert(id);
106             });
107         }
108     }
109
110     BFS_Up_Node_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
111 }
112 };
113
114 /**
115  * @brief      Partial specialization for BFS up helper for facets.
116  *
117  * @tparam     Visitor   Type of visitor functor.
118  * @tparam     Traits    Traits of the BFS traversal.
119  * @tparam     Complex   Typename of the simplicial_complex.
120  */
121 template <typename Visitor, typename Traits, typename Complex>
122 struct BFS_Up_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, Complex::topLevel> >
123 {
124     /// Simplex dimension of facets
125     static constexpr auto level = Complex::topLevel;
126     /// Typename of the current simplices
127     using CurrSimplexID = typename Complex::template SimplexID<level>;
128
129     /**
130      * @brief      Visit simplices in the current dimension and continue.
131      *
132      * @param[in]  v          Visitor functor.
133      * @param[in]  F          The simplicial_complex to traverse.
134      * @param[in]  begin      Iterator to simplices to traverse.
135      * @param[in]  end        Iterator to end of simplices to traverse.
136      *
137      * @tparam     Iterator   Typename of the iterator.
138      */
139     template <typename Iterator>
140     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
141     {
142         for (auto curr = begin; curr != end; ++curr)
143         {
144             v.visit(F, *curr);
145         }
146     }
147 };
148
149
150 /**
151  * @brief      General template for BFS down helper.
152  *
153  * @tparam     Visitor   Type of visitor functor.
154  * @tparam     Traits    Traits of the BFS traversal.
155  * @tparam     Complex   Typename of the simplicial_complex.
156  * @tparam     K          Current simplex dimension to traverse.
157  */
158 template <typename Visitor, typename Traits, typename Complex, typename K>
159 struct BFS_Down_Node {};
160
161 /**
162  * @brief      Partial specialization for BFS down helper for non facet
163  *              dimensions.
164  *
165  * @tparam     Visitor   Type of visitor functor.
166  * @tparam     Traits    Traits of the BFS traversal.
167  * @tparam     Complex   Typename of the simplicial_complex.
168  * @tparam     k          Current simplex dimension to traverse.

```

```

169 */
170 template <typename Visitor, typename Traits, typename Complex, std::size_t k>
171 struct BFS_Down_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
172 {
173     /// Simplex dimension current traversed
174     static constexpr auto level = k;
175     /// Typename of the current simplices
176     using CurrSimplexID = typename Complex::template SimplexID<level>;
177     /// Typename of boundary simplices
178     using NextSimplexID = typename Complex::template SimplexID<level-1>;
179     /// Container to use to hold boundary simplices for next recursion
180     template <typename T> using Container = typename Traits::template Container<T>;
181
182     /// Alias for the recursive call
183     using BFS_Down_Node_Next = BFS_Down_Node<Visitor, Traits, Complex,
184         std::integral_constant<std::size_t, level-1> >;
185
186     /**
187     * @brief Visit simplices in the current dimension and continue.
188     *
189     * @param[in] v Visitor functor.
190     * @param[in] F The simplicial_complex to traverse.
191     * @param[in] begin Iterator to simplices to traverse.
192     * @param[in] end Iterator to end of simplices to traverse.
193     * @tparam Iterator Typename of the iterator.
194     */
195     template <typename Iterator>
196     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
197     {
198         Container<NextSimplexID> next;
199
200         for (auto curr = begin; curr != end; ++curr)
201         {
202             if (v.visit(F, *curr))
203             {
204                 F.get_name(*curr, [&](typename Complex::KeyType a)
205                 {
206                     auto id = F.get_simplex_down(*curr, a);
207                     next.insert(id);
208                 });
209             }
210         }
211
212         BFS_Down_Node_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
213     }
214 };
215
216 /**
217 * @brief Partial specialization for BFS down helper for vertices
218 *
219 * @tparam Visitor Type of visitor functor.
220 * @tparam Traits Traits of the BFS traversal.
221 * @tparam Complex Typename of the simplicial_complex.
222 */
223 template <typename Visitor, typename Traits, typename Complex>
224 struct BFS_Down_Node<Visitor, Traits, Complex, std::integral_constant<std::size_t, 1> >
225 {
226     /**
227     * @brief Visit simplices in the current dimension and continue.
228     *
229     * @param[in] v Visitor functor.
230     * @param[in] F The simplicial_complex to traverse.
231     * @param[in] begin Iterator to simplices to traverse.
232     * @param[in] end Iterator to end of simplices to traverse.
233     * @tparam Iterator Typename of the iterator.
234     */
235     template <typename Iterator>
236     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
237     {
238         for (auto curr = begin; curr != end; ++curr)
239         {
240             v.visit(F, *curr);
241         }
242     }
243 };
244 };
245
246 /**
247 * @deprecated
248 * @brief Case to catch accidents... calling down on root is bad.
249 *
250 * @tparam Visitor { description }
251 * @tparam Traits { description }
252 * @tparam Complex { description }
253 */
254 // template <typename Visitor, typename Traits, typename Complex>

```

```

255 // struct BFS_Down_Node<Visitor, Traits, Complex,
256 // std::integral_constant<std::size_t,0>
257 // {
258 //     template <typename Iterator>
259 //     static void apply(Visitor&& v, Complex& F, Iterator begin, Iterator end)
260 //     {}
261 // };
262
263
264 /**
265  * @brief      General tempalte for BFS traversal across edges.
266  *
267  * @tparam      Visitor    Type of visitor functor.
268  * @tparam      Traits     Traits of the BFS traversal.
269  * @tparam      Complex    Typename of the simplicial_complex.
270  * @tparam      K          Current simplex dimension to traverse.
271  */
272 template <typename Visitor, typename Traits, typename Complex, typename K>
273 struct BFS_Edge {};
274
275
276 /**
277  * @brief      Partial specialization for BFS Edge for non facets.
278  *
279  * @tparam      Visitor    Type of visitor functor.
280  * @tparam      Traits     Traits of the BFS traversal.
281  * @tparam      Complex    Typename of the simplicial_complex.
282  * @tparam      k          Current simplex dimension to traverse.
283  */
284 template <typename Visitor, typename Traits, typename Complex, std::size_t k>
285 struct BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t, k> >
286 {
287     /// Current simplex dimension to traverse
288     static constexpr auto level = k;
289     /// Typename of the current EdgeID
290     using CurrEdgeID = typename Complex::template EdgeID<level>;
291     /// Typename of the next EdgeID
292     using NextEdgeID = typename Complex::template EdgeID<level+1>;
293     /// Typename of the current SimplexID
294     using CurrSimplexID = typename Complex::template SimplexID<level>;
295     /// Container to use to hold coboundary edges for next recursion.
296     template <typename T> using Container = typename Traits::template Container<T>;
297     /// Alias for the recursive call
298     using BFS_Edge_Next = BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t,
299 level+1> >;
300
301     /**
302     * @brief      Visit simplices in the current dimension and continue.
303     *
304     * @param[in]  v          Visitor functor.
305     * @param[in]  F          The simplicial_complex to traverse.
306     * @param[in]  begin      Iterator to edges to traverse.
307     * @param[in]  end        Iterator to end of edges to traverse.
308     * @tparam      Iterator  Typename of the iterator.
309     */
310     template <typename Iterator>
311     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
312     {
313         Container<NextEdgeID> next;
314         std::vector<typename Complex::KeyType> cover;
315
316         for (auto curr = begin; curr != end; ++curr)
317         {
318             v.visit(F, *curr);
319
320             CurrSimplexID n = curr->up();
321             F.get_cover(n, std::back_inserter(cover));
322             for (auto a : cover)
323             {
324                 NextEdgeID id = F.get_edge_up(n, a);
325                 next.insert(next.end(), id);
326             }
327             cover.clear();
328         }
329
330         BFS_Edge_Next::apply(std::forward<Visitor>(v), F, next.begin(), next.end());
331     }
332 };
333
334
335 /**
336  * @brief      Partial specialization for BFS Edge for facets.
337  *
338  * @tparam      Visitor    Type of visitor functor.
339  * @tparam      Traits     Traits of the BFS traversal.
340  * @tparam      Complex    Typename of the simplicial_complex.

```

```

341 */
342 template <typename Visitor, typename Traits, typename Complex>
343 struct BFS_Edge<Visitor, Traits, Complex, std::integral_constant<std::size_t, Complex::topLevel> >
344 {
345     /// Simplex dimension current traversed
346     static constexpr auto level = Complex::topLevel;
347     /// Typename of the current edge
348     using CurrEdgeID = typename Complex::template EdgeID<level>;
349
350     /**
351     * @brief Visit the edges to facets.
352     *
353     * @param[in] v Visitor functor.
354     * @param[in] F The simplicial_complex to traverse.
355     * @param[in] begin Iterator to edges to traverse.
356     * @param[in] end Iterator to end of edges to traverse.
357     *
358     * @tparam Iterator Typename of the iterator.
359     */
360     template <typename Iterator>
361     static void apply(Visitor &&v, Complex &F, Iterator begin, Iterator end)
362     {
363         for (auto curr = begin; curr != end; ++curr)
364         {
365             v.visit(F, *curr);
366         }
367     }
368 };
369
370 /// Allow repeat visits of simplices for BFS visits.
371 struct BFS_Repeat_Node_traits
372 {
373     /// Use a vector to allow duplicates
374     template <typename T> using Container = std::vector<T>;
375 };
376
377 /// No repeat traits for BFS simplex visitor.
378 struct BFS_NoRepeat_Node_Traits
379 {
380     /// Use a NodeSet to avoid duplicates
381     template <typename T> using Container = NodeSet<T>;
382 };
383
384 /// No repeat traits for BFS edge visitor.
385 struct BFS_NoRepeat_Edge_Traits
386 {
387     /// Use a NodeSet to avoid duplicates.
388     template <typename T> using Container = NodeSet<T>;
389     // template <typename Complex, typename SimplexID> auto node_next(Complex F,
390     // SimplexID s);
391 };
392 } // End namespace visitor_detail
393 /// @endcond
394
395 /**
396 * @brief Traverse BFS up the complex and apply a visitor function to each
397 * simplex visited.
398 *
399 * @param[in] v Visitor functor to apply.
400 * @param F The simplicial_complex to traverse.
401 * @param[in] s The simplex to start at.
402 *
403 * @tparam Visitor Typename of the functor.
404 * @tparam SimplexID Typename of the simplex.
405 */
406 template <typename Visitor, typename SimplexID>
407 void visit_BFS_up(Visitor &&v, typename SimplexID::complex &F, SimplexID s)
408 {
409     namespace cvd = visitor_detail;
410     cvd::BFS_Up_Node<Visitor, cvd::BFS_NoRepeat_Node_Traits, typename SimplexID::complex,
411         std::integral_constant<std::size_t, SimplexID::level> >::apply(
412         std::forward<Visitor>(v), F, &s, &s+1);
413 }
414
415 /**
416 * @brief Traverse BFS down the complex and apply a visitor function to
417 * each
418 * simplex visited.
419 *
420 * @param[in] v Visitor functor to apply.
421 * @param F The simplicial_complex to traverse.
422 * @param[in] s The simplex to start at.
423 *
424 * @tparam Visitor Typename of the functor.
425 * @tparam SimplexID Typename of the simplex.
426 */
427 template <typename Visitor, typename SimplexID>

```

```

428 void visit_BFS_down(Visitor &&v, typename SimplexID::complex &F, SimplexID s)
429 {
430     namespace cvd = visitor_detail;
431     cvd::BFS_Down_Node<Visitor, cvd::BFS_NoRepeat_Node_Traits, typename SimplexID::complex,
432         std::integral_constant<std::size_t, SimplexID::level> >::apply(
433         std::forward<Visitor>(v), F, &s, &s+1);
434 }
435
436 /**
437  * @brief      Traverse across edges BFS.
438  *
439  * @param[in]  v          Visitor functor to apply.
440  * @param      F          The simplicial_complex to traverse.
441  * @param[in]  s          The edge to start at.
442  *
443  * @tparam     Visitor    Typename of the functor.
444  * @tparam     EdgeID     Typename of the edge.
445  */
446 template <typename Visitor, typename EdgeID>
447 void edge_up(Visitor &&v, typename EdgeID::complex &F, EdgeID s)
448 {
449     namespace cvd = visitor_detail;
450     cvd::BFS_Edge<Visitor, cvd::BFS_NoRepeat_Edge_Traits, typename EdgeID::complex,
451         std::integral_constant<std::size_t, EdgeID::level> >::apply(
452         std::forward<Visitor>(v), F, &s, &s+1);
453 }
454
455
456 /**
457  * @brief      Push the immediate face neighbors into the provided iterator.
458  *
459  * This function gets the set of neighbors which share a common face. We
460  * compute this by traversing to all faces of the simplex of interest. Then we
461  * get all cofaces of this set. Depending on the type of iterator passed,
462  * duplicate simplices will be included or excluded. Note that this is the
463  * traditional definition of neighbor. For example, faces which share an edge
464  * are neighbors.
465  *
466  * @param      F          The simplicial complex
467  * @param[in]  nid        Simplex to get neighbors of.
468  * @param[in]  iter       The iterator to push members into.
469  *
470  * @tparam     Complex    Type of the simplicial complex
471  * @tparam     level      The integral level of the node
472  * @tparam     InsertIter Typename of the iterator.
473  */
474 template <class Complex, std::size_t level, class InsertIter>
475 void neighbors(Complex &F, typename Complex::template SimplexID<level> nid, InsertIter iter)
476 {
477     for (auto a : F.get_name(nid))
478     {
479         auto id = F.get_simplex_down(nid, a);
480         for (auto b : F.get_cover(id))
481         {
482             auto nbor = F.get_simplex_up(id, b);
483             if (nbor != nid)
484             {
485                 *iter++ = nbor;
486             }
487         }
488     }
489 }
490
491 /**
492  * @brief      This is a helper function to assist neighbors to automatically
493  *             deduce the integral level.
494  *
495  * @param      F          The simplicial complex.
496  * @param[in]  nid        Simplex to get neighbors of.
497  * @param[in]  iter       The iterator to push members into.
498  *
499  * @tparam     Complex    Type of the simplicial complex
500  * @tparam     level      The integral level of the node
501  * @tparam     InsertIter Typename of the iterator.
502  */
503 template <class Complex, class SimplexID, class InsertIter>
504 void neighbors(Complex &F, SimplexID nid, InsertIter iter)
505 {
506     neighbors<Complex, SimplexID::level, InsertIter>(F, nid, iter);
507 }
508
509 /**
510  * @brief      Push the immediate coface neighbors into the provided iterator.
511  *
512  * @param      F          The simplicial complex.
513  * @param[in]  nid        Simplex to get neighbors of.
514  * @param[in]  iter       The iterator to push members into.

```

```

515 *
516 * @tparam Complex      Type of the simplicial complex
517 * @tparam level        The integral level of the node
518 * @tparam InsertIter   Typename of the iterator.
519 */
520 template <class Complex, std::size_t level, class InsertIter>
521 void neighbors_up(Complex &F, typename Complex::template SimplexID<level> nid, InsertIter iter)
522 {
523     for (auto a : F.get_cover(nid))
524     {
525         auto id = F.get_simplex_up(nid, a);
526         for (auto b : F.get_name(id))
527         {
528             auto nbor = F.get_simplex_down(id, b);
529             if (nbor != nid)
530             {
531                 *iter++ = nbor;
532             }
533         }
534     }
535 }
536
537 /**
538 * @brief      This is a helper function to assist neighbors to automatically
539 *             deduce the integral level.
540 *
541 * @param      F          The simplicial complex.
542 * @param[in]  nid        Simplex to get neighbors of.
543 * @param[in]  iter       The iterator to push members into.
544 *
545 * @tparam     Complex    Type of the simplicial complex
546 * @tparam     level      The integral level of the node
547 * @tparam     InsertIter Typename of the iterator.
548 */
549 template <class Complex, class SimplexID, class InsertIter>
550 void neighbors_up(Complex &F, SimplexID nid, InsertIter iter)
551 {
552     neighbors_up<Complex, SimplexID::level, InsertIter>(F, nid, iter);
553 }
554
555
556
557 /**
558 * @brief      Code for returning a set of k-ring neighbors.
559 *
560 * @param[in]  F          The simplicial_complex to traverse.
561 * @param[in]  ring       The number of rings of neighbors to collect.
562 * @param[out] nbors      Set of previously seen simplices.
563 * @param[in]  begin      The begin
564 * @param[in]  end        The end
565 *
566 * @tparam     Complex    Typename of the simplicial_complex.
567 * @tparam     level      Simplex dimension of the simplex and neighbors.
568 * @tparam     Iterator   { description }
569 */
570 template <class Complex, std::size_t level, typename Iterator>
571 void kneighbors_up(Complex &F, int ring, std::set<typename Complex::template SimplexID<level> > &nbors, Iterator begin, Iterator end)
572 {
573     if (ring == 0)
574     {
575         return;
576     }
577     std::set<typename Complex::template SimplexID<level> > next;
578     for (auto nid = begin; nid != end; ++nid)
579     {
580         for (auto a : F.get_cover(*nid))
581         {
582             auto id = F.get_simplex_up(*nid, a);
583             for (auto b : F.get_name(id))
584             {
585                 auto nbor = F.get_simplex_down(id, b);
586                 if (nbors.insert(nbor).second)
587                 {
588                     next.insert(nbor);
589                 }
590             }
591         }
592     }
593     return kneighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
594 }
595
596
597
598 /**
599 * @brief      Helper function to help kneighbors_up to deduce the integral

```

```

602 *           level of SimplexID.
603 *
604 * @param[in] F           The simplicial complex
605 * @param[in] nid         Simplex of interest to get the nieghbors of.
606 * @param[in] ring        The number of rings to include as a neighbor.
607 * @param[out] nbors      Set of neighbors to populate.
608 *
609 * @tparam Complex        Typename of the complex.
610 * @tparam SimplexID      Typename of the SimplexID.
611 */
612 template <class Complex, class SimplexID>
613 void kneighbors_up(Complex &F, SimplexID nid, int ring, std::set<SimplexID> &nbors)
614 {
615     nbors.insert(nid);
616     std::set<SimplexID> next {
617         nid
618     };
619     kneighbors_up<Complex, SimplexID::level>(F, ring, nbors, next.begin(), next.end());
620     nbors.erase(nid);
621 }
622
623
624 /**
625 * @brief      Code for returning a set of k-ring neighbors.
626 *
627 * @param[in] F           The simplicial_complex to traverse.
628 * @param[in] ring        The number of rings of neighbors to collect.
629 * @param[out] nbors      Set of previously seen simplices.
630 * @param[in] begin       The begin
631 * @param[in] end         The end
632 *
633 * @tparam Complex        Typename of the simplicial_complex.
634 * @tparam level          Simplex dimension of the simplex and neighbors.
635 * @tparam Iterator       { description }
636 */
637 template <class Complex, std::size_t level, typename Iterator>
638 void kneighbors(Complex &F, int ring, std::set<typename Complex::template SimplexID<level> > &nbors,
639               Iterator begin, Iterator end)
640 {
641     if (ring == 0)
642     {
643         return;
644     }
645     std::set<typename Complex::template SimplexID<level> > next;
646     for (auto nid = begin; nid != end; ++nid)
647     {
648         for (auto a : F.get_name(*nid))
649         {
650             auto id = F.get_simplex_down(*nid, a);
651             for (auto b : F.get_cover(id))
652             {
653                 auto nbor = F.get_simplex_up(id, b);
654                 if (nbors.insert(nbor).second)
655                 {
656                     next.insert(nbor);
657                 }
658             }
659         }
660     }
661     return kneighbors_up<Complex, level>(F, ring-1, nbors, next.begin(), next.end());
662 }
663
664
665 /**
666 * @brief      Helper function to help kneighbors to deduce the integral
667 *             level of SimplexID.
668 *
669 * @param[in] F           The simplicial complex
670 * @param[in] nid         Simplex of interest to get the nieghbors of.
671 * @param[in] ring        The number of rings to include as a neighbor.
672 * @param[out] nbors      Set of neighbors to populate.
673 *
674 * @tparam Complex        Typename of the complex.
675 * @tparam SimplexID      Typename of the SimplexID.
676 */
677 template <class Complex, class SimplexID>
678 void kneighbors(Complex &F, SimplexID nid, int ring, std::set<SimplexID> &nbors)
679 {
680     nbors.insert(nid);
681     std::set<SimplexID> next {
682         nid
683     };
684     kneighbors<Complex, SimplexID::level>(F, ring, nbors, next.begin(), next.end());
685     nbors.erase(nid);
686 }

```



```

689
690 } // End namespace casc
691
692
693 // namespace visitor_detail
694 // {
695 // template <typename Visitor, typename Complex, std::size_t k, std::size_t
696 // ring>
697 // struct Neighbors_Up_Node
698 // {
699 //     static constexpr auto level = k;
700 //     using SimplexID = typename Complex::template SimplexID<level>;
701
702 //     using Neighbors_Up_Node_Next =
703 // Neighbors_Up_Node<Visitor, Complex, level, ring-1>;
704
705 //     template <typename Iterator>
706 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
707 // Iterator begin, Iterator end)
708 //     {
709 //         NodeSet<SimplexID> next;
710
711 //         for(auto curr = begin; curr != end; ++curr)
712 //         {
713 //             if(v.visit(F, *curr))
714 //             {
715 //                 for(auto a : F.get_cover(*curr))
716 //                 {
717 //                     auto id = F.get_simplex_up(*curr,a);
718 //                     for(auto b : F.get_name(id))
719 //                     {
720 //                         auto nbor = F.get_simplex_down(id,b);
721 //                         if(nodes.insert(nbor).second)
722 //                         {
723 //                             next.insert(nbor);
724 //                         }
725 //                     }
726 //                 }
727 //             }
728 //         }
729
730 //         Neighbors_Up_Node_Next::apply(std::forward<Visitor>(v), F, nodes,
731 // next.begin(), next.end());
732 //     }
733 // };
734
735 // template <typename Visitor, typename Complex, std::size_t k>
736 // struct Neighbors_Up_Node<Visitor, Complex, k, 0>
737 // {
738 //     static constexpr auto level = k;
739 //     using SimplexID = typename Complex::template SimplexID<level>;
740
741 //     template <typename Iterator>
742 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
743 // Iterator begin, Iterator end)
744 //     {
745 //         for(auto curr = begin; curr != end; ++curr)
746 //         {
747 //             v.visit(F, *curr);
748 //         }
749 //     }
750 // };
751
752 // template <typename Visitor, typename Complex, std::size_t k, std::size_t
753 // ring>
754 // struct Neighbors_Down_Node
755 // {
756 //     static constexpr auto level = k;
757 //     using SimplexID = typename Complex::template SimplexID<level>;
758
759 //     using Neighbors_Down_Node_Next = Neighbors_Down_Node<Visitor, Complex,
760 // level, ring-1>;
761
762 //     template <typename Iterator>
763 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
764 // Iterator begin, Iterator end)
765 //     {
766 //         NodeSet<SimplexID> next;
767
768 //         for(auto curr = begin; curr != end; ++curr)
769 //         {
770 //             if(v.visit(F, *curr))
771 //             {
772 //                 for(auto a : F.get_name(*curr))
773 //                 {
774 //                     auto id = F.get_simplex_down(*curr,a);
775 //                     for(auto b : F.get_cover(id))

```

```

776 //          {
777 //              auto nbor = F.get_simplex_up(id,b);
778 //              if(nodes.insert(nbor).second)
779 //              {
780 //                  next.insert(nbor);
781 //              }
782 //          }
783 //      }
784 //  }
785 // }
786
787 //      Neighbors_Down_Node_Next::apply(std::forward<Visitor>(v), F, nodes,
788 // next.begin(), next.end());
789 // }
790 // };
791
792 // template <typename Visitor, typename Complex, std::size_t k>
793 // struct Neighbors_Down_Node<Visitor, Complex, k, 0>
794 // {
795 //     static constexpr auto level = k;
796 //     using SimplexID = typename Complex::template SimplexID<level>;
797
798 //     template <typename Iterator>
799 //     static void apply(Visitor&& v, Complex& F, NodeSet<SimplexID>& nodes,
800 // Iterator begin, Iterator end)
801 //     {
802 //         for(auto curr = begin; curr != end; ++curr)
803 //         {
804 //             v.visit(F, *curr);
805 //         }
806 //     }
807 // };
808 // }
809
810
811 // template <std::size_t rings, typename Visitor, typename SimplexID>
812 // void visit_neighbors_up(Visitor&& v, typename SimplexID::complex& F,
813 // SimplexID s)
814 // {
815 //     NodeSet<SimplexID> nodes{s};
816 //     namespace cvd = visitor_detail;
817 //     cvd::Neighbors_Up_Node<Visitor,typename
818 // SimplexID::complex,SimplexID::level,rings>::apply(
819 //         std::forward<Visitor>(v),F,nodes,&s,&s+1);
820 // }
821
822 // template <std::size_t rings, typename Visitor, typename SimplexID>
823 // void visit_neighbors_down(Visitor&& v, typename SimplexID::complex& F,
824 // SimplexID s)
825 // {
826 //     NodeSet<SimplexID> nodes{s};
827 //     namespace cvd = visitor_detail;
828 //     cvd::Neighbors_Down_Node<Visitor,typename
829 // SimplexID::complex,SimplexID::level,rings>::apply(
830 //         std::forward<Visitor>(v),F,nodes,&s,&s+1);
831 // }

```

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.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file decimate.h
27  * @brief Meta-data aware decimation functions.
28  */
29
30 #pragma once
31
32 #include <typeinfo>
33
34 #include "SimplexSet.h"
35 #include "SimplexMap.h"
36 #include "CASCTraversals.h"
37 #include "CASCFunctions.h"
38
39 #if __has_cpp_attribute(maybe_unused)

```

```

40 #define MAYBE_UNUSED [[maybe_unused]]
41 #else
42 #define MAYBE_UNUSED
43 #endif
44
45 namespace casc
46 {
47     /// @cond detail
48     /// Namespace for decimation related helpers
49     namespace decimation_detail
50     {
51         /**
52          * @brief      A multi-vector of simplex, data pairs.
53          *
54          * @tparam      Complex  Typename of the simplicial_complex.
55          */
56         template <typename Complex>
57         struct SimplexDataSet
58         {
59             /// Typename of vertices
60             using KeyType = typename Complex::KeyType;
61
62             /**
63              * @brief      Data type makes a pair of array of keys to data type.
64              *
65              * @tparam      k      Dimension of simplex.
66              * @tparam      T      Typename of the data.
67              */
68             template <std::size_t k, typename T>
69             struct DataType
70             {
71                 /// Pair of array to type
72                 using type = std::pair<std::array<KeyType, k>, T>;
73             };
74
75             /**
76              * @brief      DataType for simplices with no data.
77              *
78              * @tparam      k      Dimension of the simplex.
79              */
80             template <std::size_t k>
81             struct DataType<k, void>
82             {
83                 /// Array of keys
84                 using type = std::array<KeyType, k>;
85             };
86
87             /// Template to resolve NodeData types for DataType.
88             template <std::size_t j>
89             using DataSet = typename DataType<j, typename Complex::template NodeData<j> >::type;
90             /// Sequence of compile time integers.
91             using LevelIndex = typename std::make_index_sequence<Complex::numLevels>;
92             /// Tuple of DataSets corresponding to an integral level.
93             using SimplexIDLevel = typename util::int_type_map<std::size_t,
94                                                         std::tuple, LevelIndex, DataSet>::type;
95
96             /// Helper vector definition for util.
97             template <class T> using vector = std::vector<T>;
98             /// Vector of DataTypes for each integral level.
99             using type = typename util::type_map<SimplexIDLevel, vector>::type;
100         };
101
102         /**
103          * @brief      Struct functional to get the complete neighborhood around a
104          *              simplex.
105          *
106          * @tparam      Complex  Type of simplicial complex
107          */
108         template <typename Complex>
109         struct GetCompleteNeighborhood
110         {
111             /// Alias for SimplexSet
112             using SimplexSet = typename casc::SimplexSet<Complex>;
113
114             /**
115              * @brief      Constructor
116              *
117              * @param      p      SimplexSet to use to pass results back
118              */
119             GetCompleteNeighborhood(SimplexSet* p) : pLevels(p) {}
120
121             /**
122              * @brief      Continue traversing, to the next level
123              *
124              * @return      True, continue the BFS
125              */
126             template <std::size_t level>
127             bool visit(Complex &, typename Complex::template SimplexID<level>)

```

```

127     {
128         return true;
129     }
130
131     /**
132     * @brief      Terminal case, go back up (visit_node_up).
133     * @param      F          Simplicial Complex
134     * @param[in]   s          Simplex of interest
135     * @return     False, stop the BFS traversal
136     */
137     bool visit(Complex &F, typename Complex::template SimplexID<1> s)
138     {
139         visit_BFS_up(
140             func_detail::SimplexAggregator<Complex>(pLevels), F, s);
141         return false;
142     }
143
144 private:
145     /// Pointer to SimplexSet to store the complete neighborhood.
146     SimplexSet* pLevels;
147 };
148
149 /**
150 * @brief      Move found simplices from pLevels to pGrab.
151 * @param      Complex      Typename of Simplicial Complex
152 */
153 template <typename Complex>
154 struct GrabVisitor
155 {
156     /// Alias for SimplexSet
157     using SimplexSet = typename csc::SimplexSet<Complex>;
158
159     /**
160     * @brief      Constructor
161     * @param      p          SimplexSet with complete neighborhood.
162     * @param      grab       SimplexSet to store grabbed simplices.
163     */
164     GrabVisitor(SimplexSet* p, SimplexSet* grab) : pLevels(p), pGrab(grab) {}
165
166     template <std::size_t level>
167     bool visit(Complex &, typename Complex::template SimplexID<level> s)
168     {
169         if (pLevels->find(s) != pLevels->template end<level>())
170         {
171             //std::cout << "GrabVisitor (found): " << s << std::endl;
172             pLevels->erase(s);
173             pGrab->insert(s);
174             return true;
175         }
176         else
177         {
178             return false;
179         }
180     }
181
182 private:
183     /// SimplexSet with the complete neighborhood
184     SimplexSet* pLevels;
185     /// SimplexSet with grabbed simplices
186     SimplexSet* pGrab;
187 };
188
189 template <typename Complex, std::size_t BaseLevel>
190 struct InnerVisitor
191 {
192     using SimplexSet = typename csc::SimplexSet<Complex>;
193     using SimplexMap = typename csc::SimplexMap<Complex>;
194     using Simplex = typename Complex::template SimplexID<BaseLevel>;
195     using KeyType = typename Complex::KeyType;
196
197     InnerVisitor(SimplexSet* p, Simplex s, KeyType np, SimplexMap* rv)
198         : pLevels(p), simplex(s), new_point(np), data(rv) {}
199
200     /**
201     * @brief      Overloaded visit function
202     * @param      F          { parameter_description }
203     * @param[in]   <unnamed> { parameter_description }
204     * @tparam      OldLevel  { description }
205     * @return     { description_of_the_return_value }
206     */

```

```

214     template <std::size_t OldLevel>
215     bool visit(Complex &F, typename Complex::template SimplexID<OldLevel> s)
216     {
217         constexpr std::size_t NewLevel = OldLevel - BaseLevel + 1;
218
219         if (pLevels->find(s) != pLevels->template end<OldLevel>())
220         {
221             //std::cout << "InnerVisitor (found): " << s << std::endl;
222             std::array<KeyType, OldLevel> old_name = F.get_name(s);
223             std::array<KeyType, BaseLevel> base_name = F.get_name(simplex);
224             using NewArrayType = std::array<KeyType, NewLevel>;
225             NewArrayType new_name;
226
227             std::size_t i = 0;    // new_name
228             std::size_t j = 0;    // old_name
229             std::size_t k = 0;    // base_name
230
231             new_name[i++] = new_point;
232
233             // Remove base_name from old_name and append to new_name
234             while (i < NewLevel)
235             {
236                 if (k >= BaseLevel) {
237                     // append to new_name and increment
238                     new_name[i++] = old_name[j++];
239                     continue;
240                 }
241                 if (base_name[k] == old_name[j])
242                 {
243                     // if equivalent than skip the value
244                     ++j; ++k;
245                 }
246                 else
247                 {
248                     // append to new_name and increment
249                     new_name[i++] = old_name[j++];
250                 }
251             }
252
253             SimplexSet grab;
254             visit_BFS_down(GrabVisitor<Complex>(pLevels, &grab), F, s);
255
256             auto &levelMap = casc::get<NewLevel>(*data);
257             auto it = levelMap.find(new_name);
258             if (it != levelMap.end())
259             {
260                 it->second.insert(grab);
261             }
262             else
263             {
264                 MAYBE_UNUSED auto ret = levelMap.insert(
265                     std::pair<NewArrayType, SimplexSet>(new_name, grab));
266                 assert(ret.second);
267             }
268         }
269         return true;
270     }
271
272 private:
273     SimplexSet* pLevels;
274     Simplex simplex;
275     KeyType new_point;
276     SimplexMap* data;
277 };
278
279
280 template <typename Complex>
281 struct MainVisitor
282 {
283     using SimplexSet = typename casc::SimplexSet<Complex>;
284     using SimplexMap = typename casc::SimplexMap<Complex>;
285     using KeyType = typename Complex::KeyType;
286
287     MainVisitor(SimplexSet* p, KeyType np, SimplexMap* rv)
288         : pLevels(p), new_point(np), data(rv) {}
289
290     template <std::size_t level>
291     bool visit(Complex &F, typename Complex::template SimplexID<level> s)
292     {
293         //std::cout << "MainVisitor: " << s << std::endl;
294         visit_BFS_up(
295             InnerVisitor<Complex, level>(
296                 pLevels, s, new_point, data),
297                 F, s);
298         return true;
299     }
300

```

```

301     private:
302         SimplexSet* pLevels;
303         KeyType     new_point;
304         SimplexMap* data;
305 };
306
307 template <typename Complex, template<typename> class Callback>
308 struct RunCallback
309 {
310     using SimplexMap = typename casc::SimplexMap<Complex>;
311     using SimplexSet = typename casc::SimplexSet<Complex>;
312     using SimplexDataSet = typename SimplexDataSet<Complex>::type;
313     using KeyType = typename Complex::KeyType;
314     template <std::size_t level>
315     using DataType = typename Complex::template NodeData<level>;
316
317     template <std::size_t k, typename ReturnType>
318     struct PerformCallback
319     {
320     {
321         static void apply(Complex &F, Callback<Complex> &&clbk,
322                         SimplexDataSet &rv,
323                         const std::array<KeyType, k> &new_name,
324                         const SimplexSet &merged)
325         {
326             ReturnType rval = clbk(F, new_name, merged);
327             std::get<k>(rv).push_back(std::make_pair(new_name, rval));
328         }
329     };
330
331     template <std::size_t k>
332     struct PerformCallback<k, void>
333     {
334         static void apply(Complex &F, Callback<Complex> &&clbk,
335                         SimplexDataSet &rv,
336                         const std::array<KeyType, k> &new_name,
337                         const SimplexSet &merged)
338         {
339             clbk(F, new_name, merged);
340             std::get<k>(rv).push_back(new_name);
341         }
342     };
343
344     template <std::size_t k>
345     static void apply(Complex &F, SimplexMap &S,
346                     Callback<Complex> &&clbk, SimplexDataSet &rv)
347     {
348     {
349         auto &levelMap = casc::get<k>(S);
350         for (auto s : levelMap)
351         {
352             PerformCallback<k, DataType<k> >::apply(F, std::forward<Callback<Complex> >(clbk),
353             rv, s.first, s.second);
354         }
355     }
356 };
357
358 template <typename Complex>
359 struct PerformRemoval
360 {
361     template <std::size_t k>
362     static void apply(Complex &F, casc::SimplexSet<Complex> &S)
363     {
364         for (auto curr : casc::get<k>(S))
365             F.remove(curr);
366     }
367 };
368
369 template <typename Complex>
370 struct PerformInsertion {
371     using KeyType = typename Complex::KeyType;
372
373     template <std::size_t k, class T>
374     static void insert(Complex &F, std::pair<std::array<KeyType, k>, T> P)
375     {
376         F.insert(P.first, P.second);
377     }
378
379     template <std::size_t k>
380     static void insert(Complex &F, std::array<KeyType, k> A)
381     {
382         F.insert(A);
383     }
384
385     template <std::size_t k>
386     static void apply(Complex
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
841
842
843
844
845
846
847
848
849
850
851
852
853
854
855
856
857
858
859
860
861
862
863
864
865
866
867
868
869
870
871
872
873
874
875
876
877
878
879
880
881
882
883
884
885
886
887
888
889
890
891
892
893
894
895
896
897
898
899
900
901
902
903
904
905
906
907
908
909
910
911
912
913
914
915
916
917
918
919
920
921
922
923
924
925
926
927
928
929
930
931
932
933
934
935
936
937
938
939
940
941
942
943
944
945
946
947
948
949
950
951
952
953
954
955
956
957
958
959
960
961
962
963
964
965
966
967
968
969
970
971
972
973
974
975
976
977
978
979
980
981
982
983
984
985
986
987
988
989
990
991
992
993
994
995
996
997
998
999

```

```

388         typename SimplexDataSet<Complex>::type &data)
389     {
390         for (auto curr : std::get<k>(data))
391         {
392             insert(F, curr);
393         }
394     }
395 };
396
397 template <typename Complex>
398 struct DoomedHelper
399 {
400     template <std::size_t k>
401     static void apply(SimplexSet<Complex> &doomed, SimplexMap<Complex> &simplexMap){
402         auto s = casc::get<k>(simplexMap);
403         for (auto map : s){
404             doomed.insert(map.second);
405         }
406     }
407 };
408 } // end namespace decimation_detail
409 /// @endcond
410
411 /**
412  * @brief      Remove simplex in SimplexSet S from complex F
413  *
414  * @param      F      The simplicial_complex to remove from.
415  * @param      S      SimplexSet of simplices to remove.
416  *
417  * @tparam      Complex  Typename of complex
418  */
419 template <typename Complex>
420 void perform_removal(Complex &F, casc::SimplexSet<Complex> &S)
421 {
422     using SimplexSet = typename casc::SimplexSet<Complex>;
423     using LevelIndex = typename SimplexSet::cRevIndex;
424     util::int_for_each<std::size_t, LevelIndex>{
425         decimation_detail::PerformRemoval<Complex>(), F, S);
426 }
427
428 /**
429  * @brief      Insert all simplices in SimplexSet 'S' into complex 'F'
430  *
431  * @param      F      The simplicial_complex to insert into.
432  * @param      S      SimplexSet of simplices to insert.
433  *
434  * @tparam      Complex  Typename of complex
435  */
436 template <typename Complex>
437 void perform_insertion(Complex &F,
438                       typename decimation_detail::SimplexDataSet<Complex>::type &S)
439 {
440     using SimplexSet = typename casc::SimplexSet<Complex>;
441     using LevelIndex = typename SimplexSet::cLevelIndex;
442     util::int_for_each<std::size_t, LevelIndex>{
443         decimation_detail::PerformInsertion<Complex>(), F, S);
444 }
445
446 /**
447  * @brief      Run the user specified callback function
448  *
449  * @param[in]   F      The simplicial_complex
450  * @param[in]   S      SimplexMap of
451  * @param[in]   clbk   User specified callback functor
452  * @param[out]  rv      Multi-vector to place results.
453  *
454  * @tparam      Complex  Typename of the simplicial_complex
455  * @tparam      Callback  Typename of the template template callback functor
456  */
457 template <typename Complex, template<typename> class Callback>
458 void run_user_callback(Complex &F,
459                       casc::SimplexMap<Complex> &S,
460                       Callback<Complex> &&clbk,
461                       typename decimation_detail::SimplexDataSet<Complex>::type &rv)
462 {
463     using SimplexMap = typename casc::SimplexMap<Complex>;
464     using LevelIndex = typename SimplexMap::cLevelIndex;
465     util::int_for_each<std::size_t, LevelIndex>{
466         decimation_detail::RunCallback<Complex, Callback>(),
467         F, S, std::forward<Callback<Complex>>(clbk), rv);
468 }
469
470 /**
471  * @brief      Decimate a simplex of any dimension while considering any

```



```

475 *           meta-data stores on decimated simplices.
476 *
477 * @param[in] F           simplicial_complex to operate on.
478 * @param[in] s           Simplex to decimate.
479 * @param[in] clbk        Callback function to map meta-data
480 *
481 * @tparam Complex        Typename of the simplicial_complex
482 * @tparam Simplex        Typename of the simplex
483 * @tparam Callback       Typename of the template template callback functor
484 */
485 template <typename Complex, typename Simplex, template<typename> class Callback>
486 void decimate(Complex &F, Simplex s, Callback<Complex> &clbk)
487 {
488     /// Alias for SimplexSet
489     using SimplexSet = typename casc::SimplexSet<Complex>;
490     /// Alias for SimplexMap
491     using SimplexMap = typename casc::SimplexMap<Complex>;
492
493     // Create the vertex to replace 's'
494     typename Complex::KeyType np = F.add_vertex();
495     SimplexSet nbhd;
496     SimplexMap simplexMap;
497
498     // Get the complete neighborhood
499     visit_BFS_down(
500         decimation_detail::GetCompleteNeighborhood<Complex>(&nbhd),
501         F, s);
502
503     SimplexSet doomed = nbhd; // Backup the neighborhood
504     // Call MainVisitor -> InnerVisitor -> GrabVisitor sequence
505     visit_BFS_down(
506         decimation_detail::MainVisitor<Complex>(&nbhd, np, &simplexMap),
507         F, s);
508
509     // Run the user specified callback
510     typename decimation_detail::SimplexDataSet<Complex>::type rv;
511     run_user_callback(F, simplexMap, std::forward<Callback<Complex>>(&clbk), rv);
512     perform_removal(F, doomed); // Remove simplices in the neighborhood
513     perform_insertion(F, rv);    // Insert new simplices
514 }
515
516 /**
517 * @brief      Given a simplex to decimate generate a pre-post mapping
518 *
519 * @param[in] F           simplicial_complex to operate on.
520 * @param[in] s           Simplex to decimate.
521 * @param      simplexMap The simplex map to populate
522 *
523 * @tparam Complex        Typename of the simplicial_complex
524 * @tparam Simplex        Typename of the simplex
525 */
526 template <typename Complex, typename Simplex>
527 typename Complex::KeyType decimateFirstHalf(Complex &F, Simplex s, SimplexMap<Complex> &simplexMap)
528 {
529     /// Alias for SimplexSet
530     using SimplexSet = typename casc::SimplexSet<Complex>;
531
532     // Create the vertex to replace 's'
533     typename Complex::KeyType np = F.add_vertex();
534     SimplexSet nbhd;
535
536     // Get the complete neighborhood
537     visit_BFS_down(
538         decimation_detail::GetCompleteNeighborhood<Complex>(&nbhd),
539         F, s);
540
541     // Call MainVisitor -> InnerVisitor -> GrabVisitor sequence
542     visit_BFS_down(
543         decimation_detail::MainVisitor<Complex>(&nbhd, np, &simplexMap),
544         F, s);
545     return np;
546 }
547
548 /**
549 * @brief      Given a simplexMap and mapped resulting data execute the
550 *             decimation.
551 *
552 * @param      F           Simplicial complex to operate on
553 * @param      simplexMap  SimplexMap mapping simplices before and after decimation
554 * @param      rv          Resulting data for each simplex
555 *
556 * @tparam Complex        Typename of the complex of interest
557 */
558 template <typename Complex>
559 void decimateBackHalf(Complex &F, SimplexMap<Complex> &simplexMap, typename
560     decimation_detail::SimplexDataSet<Complex>::type &rv){

```

```

561
562     SimplexSet<Complex> doomed;
563     util::int_for_each<std::size_t, typename
SimplexMap<Complex>::cLevelIndex>(decimation_detail::DoomedHelper<Complex>(), doomed, simplexMap);
564
565     perform_removal(F, doomed); // Remove simplices in the neighborhood
566     perform_insertion(F, rv);   // Insert new simplices
567 }
568
569 } // 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.](#)

```

1  /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  *   and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26 * @file index_tracker.h
27 * @brief B-tree based interval tracker.
28 */
29
30 #pragma once
31
32 #include <iostream>
33 #include <assert.h>
34 #include <array>
35 #include <vector>
36 #include <cstdlib>
37 #include <limits>
38
39
40 /// Index tracker namespace
41 namespace index_tracker
42 {
43     /// B-tree internal data structures
44     namespace index_tracker_detail {
45
46
47         /**
48          * @brief Interval object represents a range.
49          *
50          * @tparam T Typename of the interval data
51          */
52         template <typename T>
53         struct Interval
54         {
55             /// Default constructor
56             Interval() : _a(0), _b(0) {}
57             /// Construct an interval from a to a+1
58             Interval(T a) : _a(a), _b(a+1) {}
59             /// Construct an interval from a to b
60             Interval(T a, T b) : _a(a), _b(b) { assert(a <= b); }
61             /// Copy constructor
62             Interval(const Interval<T>& rhs) : _a(rhs._a), _b(rhs._b) {}
63
64             /**
65              * @brief Assignment operator overload.

```

```

66      *
67      * @param[in] rhs The right hand side
68      *
69      * @return Reference to this
70      */
71      Interval& operator=(const Interval& rhs)
72      {
73          _a = rhs._a;
74          _b = rhs._b;
75          return *this;
76      }
77
78      /// Is x in the bounds of the interval
79      bool has(T x) { return _a <= x && x < _b; }
80
81      /// Get the lower inclusive bound of the interval
82      T lower() const { return _a; }
83      /// Get the upper exclusive bound of the interval
84      T upper() const { return _b; }
85
86      /// Get the lower inclusive bound of the interval
87      T& lower() { return _a; }
88      /// Get the upper exclusive bound of the interval
89      T& upper() { return _b; }
90
91      /// Get the size of the interval
92      std::size_t size() { return _b - _a; }
93
94  private:
95      T _a;    /// Inclusive lower bound
96      T _b;    /// Exclusive upper bound
97  };
98
99  template <typename T>
100  bool operator<(const Interval<T>& x, const Interval<T>& y)
101  {
102      return x.upper() <= y.lower();
103  }
104
105  template <typename T>
106  bool operator>(const Interval<T>& x, const Interval<T>& y)
107  {
108      return x.lower() >= y.upper();
109  }
110
111  template <typename T>
112  bool operator<(T x, const Interval<T>& y)
113  {
114      return x < y.lower();
115  }
116
117  template <typename T>
118  bool operator>(const Interval<T>& x, T y)
119  {
120      return x.lower() > y;
121  }
122
123  template <typename T>
124  bool operator<(const Interval<T>& x, T y)
125  {
126      return x.upper() <= y;
127  }
128
129  template <typename T>
130  bool operator>(T x, const Interval<T>& y)
131  {
132      return x >= y.upper();
133  }
134
135  template <typename T>
136  bool operator==(const Interval<T>& x, const Interval<T>& y)
137  {
138      return (x.lower() == y.lower()) && (x.upper() && y.upper());
139  }
140
141  template <typename T>
142  std::ostream& operator<<(std::ostream& out, const Interval<T>& x)
143  {
144      out << "[" << x.lower() << "~" << x.upper() << ")";
145      return out;
146  }
147
148  template <typename T>
149  int merge(Interval<T>& A, T x)
150  {
151      // If x isn't the next lower value return 0
152      if(x + 1 < A.lower())

```

```

153         return 0;
154     else if(x + 1 == A.lower())
155     {
156         // if x is the next lowest value assign to lower
157         A.lower() = x;
158         return 1;
159     }
160     else if(A.lower() <= x && x < A.upper()) // x is in range already
161         return 2;
162     else if(A.upper() == x)
163     {
164         // x is the next higher assign upper
165         A.upper() = x + 1;
166         return 3;
167     }
168     else if(A.upper() < x)
169         // x isn't next after this range return 4
170         return 4;
171     else
172         return 5; // Something undefined happened.
173 }
174
175
176 /**
177  * @brief      An array based BTree
178  *
179  * @tparam     _T      { description }
180  * @tparam     _d      { description }
181  */
182 template <typename _T, std::size_t _d>
183 struct BTreeNode
184 {
185     static constexpr std::size_t d = _d;
186     static constexpr std::size_t N = 2*d+1;
187     using Scalar = _T;
188     using Data = Interval<Scalar>;
189     using Pointer = BTreeNode*;
190
191     BTreeNode() {}
192     BTreeNode(const Data& t)
193         : k(1), next{nullptr, nullptr}
194     {
195         data[0] = t;
196     }
197
198     template <typename Iter>
199     BTreeNode(Iter begin, Iter end)
200     {
201         k = 0;
202         while(begin != end)
203         {
204             next[k] = nullptr;
205             data[k++] = *begin++;
206         }
207         next[k] = nullptr;
208     }
209
210     std::size_t k;
211     std::array<Data, N> data;
212     std::array<Pointer, N+1> next;
213 };
214
215 template <typename Node> using Pointer = typename Node::Pointer;
216 template <typename Node> using Data = typename Node::Data;
217 template <typename Node> using Scalar = typename Node::Scalar;
218
219
220
221 template <typename Node>
222 void rebalance(Pointer<Node> head, std::size_t i)
223 {
224     Pointer<Node> curr = head->next[i];
225
226     if(curr->k == Node::N)
227     {
228         // Pointer<Node> left = curr; // UNUSED
229         Pointer<Node> right = new Node(curr->data.begin() + Node::d + 1, curr->data.end());
230         curr->k = Node::d;
231
232         if(curr->next[0] == nullptr)
233         {
234             right->next[0] = nullptr;
235         }
236         else
237         {
238             for(std::size_t i = 0; i <= Node::d; ++i)
239             {

```

```

240         right->next[i] = curr->next[Node::d + i + 1];
241     }
242 }
243
244 Data<Node> up = curr->data[Node::d];
245
246 for(std::size_t j = head->k; j > i; --j)
247 {
248     head->data[j] = head->data[j-1];
249     head->next[j+1] = head->next[j];
250 }
251 head->data[i] = up;
252 head->next[i+1] = right;
253 ++(head->k);
254 }
255 else if(curr->k < Node::d)
256 {
257     if(i > 0 && head->next[i-1]->k > Node::d)
258     {
259         Pointer<Node> left = head->next[i-1];
260         Pointer<Node> right = head->next[i];
261
262         if(right->next[0] != nullptr)
263             right->next[right->k + 1] = right->next[right->k];
264         for(std::size_t j = right->k; j > 0; --j)
265         {
266             right->data[j] = right->data[j-1];
267             if(left->next[0] != nullptr)
268                 right->next[j] = right->next[j-1];
269         }
270         right->data[0] = head->data[i-1];
271         if(left->next[0] != nullptr)
272             right->next[0] = left->next[left->k];
273         ++(right->k);
274
275         head->data[i-1] = left->data[left->k-1];
276
277         --(left->k);
278
279         // std::cout << "Rotate Right" << std::endl;
280     }
281     else if(i < head->k && head->next[i+1]->k > Node::d)
282     {
283         Pointer<Node> left = head->next[i];
284         Pointer<Node> right = head->next[i+1];
285
286         left->data[left->k] = head->data[i];
287         ++(left->k);
288         if(left->next[0] != nullptr)
289             left->next[left->k] = right->next[0];
290
291         head->data[i] = right->data[0];
292         for(std::size_t j = 0; j < right->k - 1; ++j)
293         {
294             right->data[j] = right->data[j+1];
295             if(right->next[0] != nullptr)
296                 right->next[j] = right->next[j+1];
297         }
298         --(right->k);
299         if(right->next[0] != nullptr)
300             right->next[right->k] = right->next[right->k + 1];
301
302         // std::cout << "Rotate Left" << std::endl;
303     }
304     else
305     {
306         if(i < head->k)
307         {
308             Pointer<Node> left = head->next[i];
309             Pointer<Node> right = head->next[i+1];
310
311             left->data[(left->k)++] = head->data[i];
312             for(std::size_t j = 0; j < right->k; ++j)
313             {
314                 left->data[left->k] = right->data[j];
315                 if(left->next[0] != nullptr)
316                     left->next[left->k] = right->next[j];
317                 ++(left->k);
318             }
319             if(left->next[0] != nullptr)
320                 left->next[left->k] = right->next[right->k];
321
322             delete right;
323
324             --(head->k);
325             for(std::size_t j = i; j < head->k; ++j)
326                 {

```

```

327         head->data[j] = head->data[j+1];
328         head->next[j+1] = head->next[j+2];
329     }
330 }
331 else
332 {
333     Pointer<Node> left = head->next[i-1];
334     Pointer<Node> right = head->next[i];
335
336     left->data[left->k] = head->data[i-1];
337     ++(left->k);
338     for(std::size_t j = 0; j < right->k; ++j)
339     {
340         left->data[left->k] = right->data[j];
341         if(left->next[0] != nullptr)
342             left->next[left->k] = right->next[j];
343         ++(left->k);
344     }
345     if(left->next[0] != nullptr)
346         left->next[left->k] = right->next[right->k];
347
348     delete right;
349     --(head->k);
350 }
351 }
352 }
353 }
354
355 template <typename Node>
356 void insert_H(Pointer<Node> head, const Data<Node>& data)
357 {
358     if(head->next[0] == nullptr)
359     {
360         const auto k = head->k;
361
362         std::size_t i = 0;
363         while(i < k && head->data[i] < data)
364         {
365             ++i;
366         }
367         for(std::size_t j = k; j > i; --j)
368         {
369             head->data[j] = head->data[j-1];
370         }
371         head->data[i] = data;
372         head->k = k+1;
373     }
374     else
375     {
376         const auto k = head->k;
377
378         std::size_t i = 0;
379         while(i < k && head->data[i] < data)
380             ++i;
381
382         insert_H<Node>(head->next[i], data);
383         rebalance<Node>(head, i);
384     }
385 }
386
387 template <typename Node>
388 Pointer<Node> insert(Pointer<Node> head, Data<Node> data)
389 {
390     if(head == nullptr)
391     {
392         return new Node(data);
393     }
394     else
395     {
396         insert_H<Node>(head, data);
397         if(head->k == Node::N)
398         {
399             Pointer<Node> nn = new Node();
400             nn->k = 0;
401             nn->next[0] = head;
402             rebalance<Node>(nn, 0);
403             return nn;
404         }
405         else
406         {
407             return head;
408         }
409     }
410 }
411
412 template <typename Node>
413 bool get(Pointer<Node> head, Data<Node> data)

```



```

414     {
415         if(head->next[0] == nullptr)
416         {
417             for(std::size_t i = 0; i < head->k; ++i)
418             {
419                 if(data == head->data[i])
420                 {
421                     return true;
422                 }
423             }
424             return false;
425         }
426         else
427         {
428             for(std::size_t i = 0; i < head->k; ++i)
429             {
430                 if(data < head->data[i])
431                 {
432                     return get<Node>(head->next[i], data);
433                 }
434                 else if(data == head->data[i])
435                 {
436                     return true;
437                 }
438             }
439             return get<Node>(head->next[head->k], data);
440         }
441     }
442 }
443
444
445 template <typename Node>
446 void get_replacement(Pointer<Node> head, Data<Node>& key)
447 {
448     if(head->next[0] == nullptr)
449     {
450         key = head->data[head->k-1];
451         --(head->k);
452     }
453     else
454     {
455         get_replacement<Node>(head->next[head->k], key);
456         rebalance<Node>(head, head->k);
457     }
458 }
459
460 template <typename Node>
461 void remove_H(Pointer<Node> head, Data<Node> data)
462 {
463     if(head->next[0] == nullptr)
464     {
465         for(std::size_t i = 0; i < head->k; ++i)
466         {
467             if(data == head->data[i])
468             {
469                 for(std::size_t j = i+1; j < head->k; ++j)
470                 {
471                     head->data[j-1] = head->data[j];
472                 }
473                 --(head->k);
474                 break;
475             }
476         }
477     }
478     else
479     {
480         for(std::size_t i = 0; i < head->k; ++i)
481         {
482             if(data < head->data[i])
483             {
484                 remove_H<Node>(head->next[i], data);
485                 rebalance<Node>(head, i);
486                 return;
487             }
488             else if(data == head->data[i])
489             {
490                 get_replacement<Node>(head->next[i], head->data[i]);
491                 rebalance<Node>(head, i);
492                 return;
493             }
494         }
495         remove_H<Node>(head->next[head->k], data);
496         rebalance<Node>(head, head->k);
497     }
498 }
499
500 template <typename Node>

```

```

501     Pointer<Node> remove(Pointer<Node> head, Data<Node> data)
502     {
503         remove_H<Node>(head, data);
504
505         if(head->k == 0)
506         {
507             Pointer<Node> rval = head->next[0];
508             delete head;
509             return rval;
510         }
511         else
512         {
513             return head;
514         }
515     }
516
517     template <typename Node>
518     void fill_left(Pointer<Node> head, Data<Node>& x)
519     {
520         if(head->next[0] == nullptr)
521         {
522             Data<Node>& left = head->data[head->k-1];
523             if(left.upper() == x.lower())
524             {
525                 x.lower() = left.lower();
526                 --(head->k);
527             }
528         }
529         else
530         {
531             fill_left<Node>(head->next[head->k], x);
532             rebalance<Node>(head, head->k);
533         }
534     }
535 }
536
537     template <typename Node>
538     void fill_right(Pointer<Node> head, Data<Node>& x)
539     {
540         if(head->next[0] == nullptr)
541         {
542             Data<Node>& right = head->data[0];
543             if(right.lower() == x.upper())
544             {
545                 x.upper() = right.upper();
546                 --(head->k);
547                 for(std::size_t i = 0; i < head->k; ++i)
548                 {
549                     head->data[i] = head->data[i+1];
550                 }
551             }
552         }
553         else
554         {
555             fill_right<Node>(head->next[0], x);
556             rebalance<Node>(head, 0);
557         }
558     }
559 }
560
561     template <typename Node>
562     void insert_scalar_H(Pointer<Node> head, Scalar<Node> data)
563     {
564         // If the
565         if(head->next[0] == nullptr)
566         {
567             const auto k = head->k;
568
569             std::size_t i;
570             for(i = 0; i < k; ++i)
571             {
572                 Data<Node>& A = head->data[i];
573                 Scalar<Node> x = data;
574
575                 if(x + 1 < A.lower())
576                 {
577                     for(std::size_t j = k; j > i; --j)
578                     {
579                         head->data[j] = head->data[j-1];
580                     }
581                     head->data[i] = data;
582                     ++(head->k);
583                     return;
584                 }
585             }
586             else if(x + 1 == A.lower())
587             {

```

```

588         A.lower() = x;
589         return;
590     }
591     else if(A.lower() <= x && x < A.upper())
592     {
593         return;
594     }
595     else if(A.upper() == x)
596     {
597         if(i + 1 < k)
598         {
599             Data<Node>& B = head->data[i+1];
600             if(x + 1 == B.lower())
601             {
602                 A.upper() = B.upper();
603                 for(std::size_t j = i+1; j < k-1; ++j)
604                 {
605                     head->data[j] = head->data[j+1];
606                 }
607                 --(head->k);
608             }
609             else
610             {
611                 A.upper() = x + 1;
612             }
613         }
614         else
615         {
616             A.upper() = x + 1;
617         }
618         return;
619     }
620 }
621 head->data[i] = data;
622 ++(head->k);
623 }
624 else
625 {
626     const auto k = head->k;
627
628     std::size_t i;
629     for(i = 0; i < k; ++i)
630     {
631         Data<Node>& A = head->data[i];
632         Scalar<Node> x = data;
633
634         if(x + 1 < A.lower())
635         {
636             insert_scalar_H<Node>(head->next[i], data);
637             rebalance<Node>(head, i);
638             return;
639         }
640         else if(x + 1 == A.lower())
641         {
642             A.lower() = x;
643             fill_left<Node>(head->next[i], A);
644             rebalance<Node>(head, i);
645             return;
646         }
647         else if(A.lower() <= x && x < A.upper())
648         {
649             return;
650         }
651         else if(A.upper() == x)
652         {
653             A.upper() = x + 1;
654             fill_right<Node>(head->next[i+1], A);
655             rebalance<Node>(head, i+1);
656             return;
657         }
658     }
659     insert_scalar_H<Node>(head->next[i], data);
660     rebalance<Node>(head, i);
661 }
662 }
663
664 template <typename Node>
665 Pointer<Node> insert_scalar(Pointer<Node> head, Scalar<Node> data)
666 {
667     if(head == nullptr)
668     {
669         return new Node(data);
670     }
671     else
672     {
673         insert_scalar_H<Node>(head, data);
674         if(head->k == Node::N)

```

```

675         {
676             Pointer<Node> nn = new Node();
677             nn->k = 0;
678             nn->next[0] = head;
679             rebalance<Node>(nn, 0);
680             return nn;
681         }
682         else if(head->k == 0)
683         {
684             Pointer<Node> rval = head->next[0];
685             delete head;
686             return rval;
687         }
688         else
689         {
690             return head;
691         }
692     }
693 }
694
695 template <typename Node>
696 void insert_left(Pointer<Node> head, const Data<Node>& x)
697 {
698     if(head->next[0] == nullptr)
699     {
700         head->data[head->k] = x;
701         ++(head->k);
702     }
703     else
704     {
705         insert_left<Node>(head->next[head->k], x);
706         rebalance<Node>(head, head->k);
707     }
708 }
709
710
711 template <typename Node>
712 bool remove_scalar_H(Pointer<Node> head, Scalar<Node> x)
713 {
714     if(head->next[0] == nullptr)
715     {
716         const auto k = head->k;
717
718         std::size_t i;
719         for(i = 0; i < k; ++i)
720         {
721             Data<Node>& A = head->data[i];
722
723             if(x < A.lower())
724             {
725                 std::cout << "if(x < A.lower())" << std::endl;
726                 return false;
727             }
728             else if(x == A.lower())
729             {
730                 std::cout << "if(x == A.lower())" << std::endl;
731                 if(x + 1 == A.upper())
732                 {
733                     std::cout << "if(x + 1 == A.upper())" << std::endl;
734                     --(head->k);
735                     for(std::size_t j = i; j < head->k; ++j)
736                     {
737                         head->data[j] = head->data[j+1];
738                     }
739                     return true;
740                 }
741                 A.lower() = x + 1;
742                 return true;
743             }
744             else if(*A.lower() < x &&* / x + 1 < A.upper())
745             {
746                 std::cout << "x + 1 < A.upper()" << std::endl;
747                 for(std::size_t j = head->k; j > i; --j)
748                 {
749                     head->data[j] = head->data[j-1];
750                 }
751                 ++(head->k);
752                 A.upper() = x;
753                 head->data[i+1].lower() = x + 1;
754                 return true;
755             }
756             else if(x + 1 == A.upper())
757             {
758                 std::cout << "x + 1 < A.upper()" << std::endl;
759                 A.upper() = x;
760                 return true;
761             }

```

```

762     }
763     return false;
764 }
765 else
766 {
767     const auto k = head->k;
768
769     std::size_t i;
770     for(i = 0; i < k; ++i)
771     {
772         Data<Node>& A = head->data[i];
773
774         if(x < A.lower())
775         {
776             bool rval = remove_scalar_H<Node>(head->next[i], x);
777             rebalance<Node>(head, i);
778             return rval;
779         }
780         else if(x == A.lower())
781         {
782             if(x + 1 == A.upper())
783             {
784                 get_replacement<Node>(head->next[i], A);
785                 rebalance<Node>(head, i);
786                 return true;
787             }
788             A.lower() = x + 1;
789             return true;
790         }
791         else if(*A.lower() < x &&*/ x + 1 < A.upper())
792         {
793             Data<Node> B(A.lower(), x);
794             A.lower() = x + 1;
795             insert_left<Node>(head->next[i], B);
796             rebalance<Node>(head, i);
797             return true;
798         }
799         else if(x + 1 == A.upper())
800         {
801             A.upper() = x;
802             return true;
803         }
804     }
805     bool rval = remove_scalar_H<Node>(head->next[i], x);
806     rebalance<Node>(head, i);
807     return rval;
808 }
809 }
810
811 template <typename Node>
812 bool remove_scalar(Pointer<Node>& head, Scalar<Node> data)
813 {
814     if(head == nullptr)
815     {
816         return false;
817     }
818
819     bool rval = remove_scalar_H<Node>(head, data);
820
821     if(head->k == Node::N)
822     {
823         Pointer<Node> nn = new Node();
824         nn->k = 0;
825         nn->next[0] = head;
826         rebalance<Node>(nn, 0);
827         head = nn;
828     }
829     else if(head->k == 0)
830     {
831         Pointer<Node> tmp = head;
832         head = head->next[0];
833         delete tmp;
834     }
835
836     return rval;
837 }
838
839 template <typename Node>
840 Scalar<Node> pop_scalar(Pointer<Node>& head)
841 {
842     if(head)
843     {
844         Scalar<Node> x = head->data[0].lower();
845         remove_scalar<Node>(head, x);
846         return x;
847     }
848     exit(-1);

```

```

849     }
850
851     template <typename Node>
852     void destruct(Pointer<Node> head)
853     {
854         if(head == nullptr)
855         {
856             return;
857         }
858         else
859         {
860             if(head->next[0] != nullptr)
861             {
862                 for(std::size_t i = 0; i < head->k; ++i)
863                 {
864                     destruct<Node>(head->next[i]);
865                 }
866             }
867             delete head;
868         }
869     }
870
871
872     template <typename Node>
873     Data<Node> check_order(Pointer<Node> head, Data<Node> curr)
874     {
875         if(head != nullptr)
876         {
877             if(head->next[0] == nullptr)
878             {
879                 for(std::size_t i = 0; i < head->k; ++i)
880                 {
881                     if(curr > head->data[i])
882                     {
883                         std::cout << "ORDER WRONG!!! -- " << curr << " > " << head->data[i] << std::endl;
884                         exit(1);
885                     }
886                     curr = head->data[i];
887                 }
888             }
889             else
890             {
891                 for(std::size_t i = 0; i < head->k; ++i)
892                 {
893                     curr = check_order<Node>(head->next[i], curr);
894                     if(curr > head->data[i])
895                     {
896                         std::cout << "ORDER WRONG!!! -- " << curr << " > " << head->data[i] << std::endl;
897                         exit(1);
898                     }
899                     curr = head->data[i];
900                 }
901                 curr = check_order<Node>(head->next[head->k], curr);
902             }
903         }
904         return curr;
905     }
906 } // End namespace index_tracker_detail
907
908 template <typename T, std::size_t d>
909 std::ostream& operator<<(std::ostream& out, const index_tracker_detail::BTreeNode<T,d>* head)
910 {
911     if(head == nullptr)
912     {
913         out << "[nil]";
914     }
915     else
916     {
917         out << "[";
918         for(std::size_t i = 0; i < head->k; ++i)
919         {
920             if(head->next[0] != nullptr)
921                 out << head->next[i] << " ";
922             out << head->data[i] << " ";
923         }
924         if(head->next[0] != nullptr)
925             out << head->next[head->k];
926         out << "]";
927     }
928     return out;
929 }
930
931 /**
932  * @brief      Tracker of available indices implemented as a B-tree of intervals.
933  *
934  * @tparam     _T      Typename of the indices
935  * @tparam     _d      Max number of interval bins = 2*value+1

```

```

936 */
937 template <typename _T, std::size_t _d = 16>
938 class index_tracker
939 {
940 public:
941     /// Typedef of BTree Node
942     using Node = index_tracker_detail::BTreeNode<_T, _d>;
943     using T = _T; /// Typename of the type to store
944     constexpr static std::size_t d = _d; /// Number of bins
945
946     /**
947      * @brief      Initialize with interval [0~max)
948      */
949     index_tracker()
950         : head(new Node(index_tracker_detail::Interval<T>(0, std::numeric_limits<T>::max())))
951     {}
952     ~index_tracker()
953     {
954         index_tracker_detail::destruct<Node>(head);
955     }
956
957     void insert(T x)
958     {
959         head = index_tracker_detail::insert_scalar<Node>(head, x);
960     }
961
962     index_tracker_detail::Scalar<Node> pop()
963     {
964         auto x = index_tracker_detail::pop_scalar<Node>(head);
965         return x;
966     }
967
968     void remove(index_tracker_detail::Scalar<Node> x)
969     {
970         index_tracker_detail::remove_scalar<Node>(head, x);
971     }
972
973     bool empty() const
974     {
975         return head == nullptr;
976     }
977
978     friend std::ostream& operator<<(std::ostream& out, const index_tracker& x)
979     {
980         out << x.head;
981         return out;
982     }
983
984 private:
985     index_tracker_detail::Pointer<Node> head;
986 };
987 } // 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.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file Orientable.h
27  * @brief Data type for orientability
28  */
29
30
31 #pragma once
32
33 #include <iostream>
34 #include <queue>
35 #include <set>
36
37
38 namespace casc{
39 /**
40  * @brief Class representing the orientation.
41  */
42 struct Orientable {
43     /// Integer representing +/- 1 orientation.
44     int orientation;
45 };
46
47 /// @cond detail
48 /// Namespace for orientation helpers
49 namespace orientation_detail{
50
51 template <class Complex, class SizeT >
52 struct init_orientation_helper {};
53
54 template <class Complex, std::size_t k >
55 struct init_orientation_helper<Complex, std::integral_constant<std::size_t, k>

```



```

56 {
57     static void f(Complex& F)
58     {
59         for(auto curr : F.template get_level_id<k>())
60         {
61             for(auto a : F.get_cover(curr))
62             {
63                 int orient = 1;
64                 for(auto b : F.get_name(curr))
65                 {
66                     // Count the number of indices > name
67                     if(a > b)
68                     {
69                         orient *= -1;
70                     }
71                     else
72                     {
73                         break;
74                     }
75                 }
76                 (*F.get_edge_up(curr,a)).orientation = orient;
77             }
78         }
79     }
80     init_orientation_helper<Complex, std::integral_constant<std::size_t, k+1>::f(F);
81 }
82 };
83
84 /**
85  * @brief      Terminating case for initializing orientation
86  *
87  * @tparam      Complex  Typename of the simplicial complex
88  */
89 template <typename Complex>
90 struct init_orientation_helper<Complex, std::integral_constant<std::size_t, Complex::topLevel>
91 {
92     static void f(Complex&) {}
93 };
94 } // end namespace orientation_detail
95 /// @endcond
96
97 /**
98  * @brief      Initialize the partial ordering of the simplex edges
99  *
100  * @param      F          Simplicial complex of interest
101  *
102  * @tparam      Complex  Typename of the simplicial complex
103  */
104 template <typename Complex>
105 void init_orientation(Complex& F)
106 {
107     orientation_detail::init_orientation_helper<Complex, std::integral_constant<std::size_t, 0>::f(F);
108 }
109
110 /**
111  * @brief      Clear the orientation of the facets
112  *
113  * @param      F          Simplicial complex of interest
114  *
115  * @tparam      Complex  Typename of the simplicial complex
116  */
117 template <typename Complex>
118 void clear_orientation(Complex& F)
119 {
120     // clear orientation
121     for(auto& curr : F.template get_level<Complex::topLevel>())
122     {
123         curr.orientation = 0;
124     }
125 }
126
127 // TODO: Implement this as a disjoint set operation during insertion (2)
128 /**
129  * @brief      Initializes and calculates the orientation of a
130  *              simplicial_complex.
131  *
132  * @param      F          Simplicial_complex
133  *
134  * @tparam      Complex  Typename of the simplicial_complex.
135  *
136  * @return      A tuple of the number of connected components, where the complex
137  *              is orientable, and if it is psuedo manifold.
138  */
139 template <typename Complex>
140 std::tuple<int, bool, bool> compute_orientation(Complex& F)
141 {
142     init_orientation(F);

```

```

143     clear_orientation(F);
144     return check_orientation(F);
145 }
146
147
148 /**
149  * @brief      Checks for self consistent orientation and fill in missing
150  *             orientations
151  *
152  * @param      F      Simplicial_complex
153  *
154  * @tparam     Complex Typename of the simplicial_complex.
155  *
156  * @return     A tuple of the number of connected components, where the complex
157  *             is orientable, and if it is psuedo manifold.
158  */
159 template <typename Complex>
160 std::tuple<int, bool, bool> check_orientation(Complex& F)
161 {
162     // compute orientation
163     constexpr std::size_t k = Complex::topLevel - 1;
164
165     std::deque<typename Complex::template SimplexID<k> > frontier;
166     std::set<typename Complex::template SimplexID<k> > visited;
167     int connected_components = 0;
168     bool orientable = true;
169     bool psuedo_manifold = true;
170     for(auto outer : F.template get_level_id<k>())
171     {
172         if(visited.find(outer) == visited.end())
173         {
174             ++connected_components;
175             frontier.push_back(outer);
176
177             while(!frontier.empty())
178             {
179                 typename Complex::template SimplexID<k> curr = frontier.front();
180                 if(visited.find(curr) == visited.end())
181                 {
182                     visited.insert(curr);
183
184                     auto w = F.get_cover(curr);
185
186                     if(w.size() == 1)
187                     {
188                         // w is a boundary
189                         //std::cout << curr << ":" << w[0] << " ~ Boundary" << std::endl;
190                     }
191                     else if(w.size() == 2)
192                     {
193                         auto& edge0 = *F.get_edge_up(curr, w[0]);
194                         auto& edge1 = *F.get_edge_up(curr, w[1]);
195
196                         auto& node0 = *F.get_simplex_up(curr, w[0]);
197                         auto& node1 = *F.get_simplex_up(curr, w[1]);
198
199                         // If node0 doesn't have an orientation yet... Assign one
200                         if(node0.orientation == 0)
201                         {
202                             if(node1.orientation == 0)
203                             {
204                                 node0.orientation = -1;
205                                 node1.orientation = -edge1.orientation * edge0.orientation *
206 node0.orientation;
207                             }
208                             else
209                             {
210                                 node0.orientation = -edge0.orientation * edge1.orientation *
211 node1.orientation;
212                             }
213                         }
214                         else
215                         {
216                             // if node1 doesn't have an orientation...
217                             if(node1.orientation == 0)
218                             {
219                                 node1.orientation = -edge1.orientation * edge0.orientation *
220 node0.orientation;
221                             }
222                             else
223                             {
224                                 // Check if the orientations are consistent
225                                 if(edge0.orientation*node0.orientation +
edge1.orientation*node1.orientation != 0)
226                                 {
227                                     orientable = false;
228                                     // std::cout << "++++" << std::endl;

```

```

226                                     // std::cout << edge0.orientation << " : " << node0.orientation <<
std::endl;
227                                     // std::cout << edge1.orientation << " : " << node1.orientation <<
std::endl;
228
229                                     // std::cout << " : "
230                                     // << edge0.orientation*node0.orientation +
edge1.orientation*node1.orientation
231                                     // << std::endl;
232                                     // std::cout << "-----"
233                                     // << std::endl;
234                                     // std::cout << "Non-Orientable: "
235                                     // << edge0.orientation*node0.orientation +
edge1.orientation*node1.orientation
236                                     // << std::endl;
237                                     }
238                                     }
239                                     }
240                                     neighbors_up(F, curr, std::back_inserter(frontier));
241                                     }
242                                     else
243                                     {
244                                         // W.size() != 1 or 2
245                                         psuedo_manifold = false;
246                                     }
247                                     }
248                                     frontier.pop_front();
249                                     }
250                                     }
251                                     }
252                                     return std::make_tuple(connected_components, orientable, psuedo_manifold);
253     }
254 } // 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.](#)

```

1  /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  *   and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26 * @file SimplexMap.h
27 * @brief SimplexMap data structure and associated convenience functions.
28 */
29
30 #pragma once
31
32 #include <array>
33 #include <map>
34 #include "util.h"
35 #include "stringutil.h"
36
37 namespace casc
38 {
39
40 /**
41 * @brief A multimap to represent a map of simplex indices to a set of
42 *        simplices.
43 *
44 * @tparam Complex Typename of the simplicial_complex.
45 */
46 template <typename Complex>
47 struct SimplexMap
48 {
49     /// Alias for SimplexID
50     template <std::size_t j>
51     using SimplexID = typename Complex::template SimplexID<j>;
52     /// Index sequence of types from the simplicial_complex
53     using LevelIndex = typename Complex::LevelIndex;
54     /// Index sequence starting at 1
55     using cLevelIndex = typename util::remove_first_val<std::size_t,
56                                                         LevelIndex>::type;
57     /// Reversed Index sequence
58     using RevIndex = typename util::reverse_sequence<std::size_t,
59                                                         LevelIndex>::type;
60     /// Reversed index sequence stops at 1
61     using cRevIndex = typename util::reverse_sequence<std::size_t,
62                                                         cLevelIndex>::type;
63     /// Typename of this object
64     using type_this = SimplexMap<Complex>;
65
66     /**
67     * @brief Default constructor.
68     */
69     SimplexMap() {};
70
71     // TODO: Put in convenience functions for easy accession etc... (0)
72     /**
73     * @brief Get the map for a particular simplex dimension.
74     *
75     * @tparam k Simplex dimension to retrieve.
76     *
77     * @return A map of SimplexID<k> to SimplexSet.
78     */
79     template <std::size_t k>
80     inline auto &get()
81     {
82         return std::get<k>(tupleMap);
83     }
84

```

```

83     }
84
85     /**
86      * @overload
87      */
88     template <std::size_t k>
89     inline auto &get() const
90     {
91         return std::get<k>(tupleMap);
92     }
93
94     /**
95      * @brief      Print the SimplexMap.
96      *
97      * @param      output    Handle to the stream to print to.
98      * @param[in]  S          SimplexMap to print.
99      *
100     * @return      Handle to the stream.
101     */
102     friend std::ostream &operator<<(std::ostream &output, const SimplexMap<Complex> &S)
103     {
104         output << "SimplexMap(";
105         util::int_for_each<std::size_t, LevelIndex>(PrintHelper(),
106                                                     output, S);
107         output << ")";
108         return output;
109     }
110
111     private:
112         /**
113          * @brief      Helper struct to print the SimplexMap.
114          */
115         struct PrintHelper
116         {
117             /**
118              * @brief      Print the SimplexMap.
119              *
120              * @param      output    Handle to the stream to print to.
121              * @param[in]  S          SimplexMap to print.
122              *
123              * @tparam      k          The simplex dimension to print.
124              */
125             template <std::size_t k>
126             static void apply(std::ostream &output, const SimplexMap<Complex> &S)
127             {
128                 output << "[l=" << k;
129                 auto s = std::get<k>(S.tupleMap);
130                 for (auto simplex : s)
131                 {
132                     output << ", " << to_string(simplex.first) << ":" << simplex.second;
133                 }
134                 output << "]";
135             }
136         };
137
138         /// Alias to create an Array of size k to store keys.
139         template <std::size_t k> using array = std::array<typename Complex::KeyType, k>;
140         /// A tuple of arrays of increasing size.
141         using ArrayLevel = typename util::int_type_map<std::size_t, std::tuple, LevelIndex,
142 array>::type;
143         /// Alias for a Map of type T to a SimplexSet.
144         template <class T> using map = std::map<T, SimplexSet<Complex> >;
145         /// The full tuple of maps of an Array of keys to SimplexSet.
146         typename util::type_map<ArrayLevel, map>::type tupleMap;
147
148     /**
149      * @brief      Get the map for a simplex dimension.
150      *
151      * @param      S          SimplexMap to retrieve from.
152      *
153      * @tparam      k          Simplex dimension.
154      * @tparam      Complex    Typename of the complex.
155      *
156      * @return      Returns a map of std::Array<KeyType, k> to SimplexSet.
157      */
158     template <std::size_t k, typename Complex>
159     static inline auto &get(SimplexMap<Complex> &S)
160     {
161         return S.template get<k>();
162     }
163
164     /// @overload
165     template <std::size_t k, typename Complex>
166     static inline auto &get(const SimplexMap<Complex> &S)
167     {
168         return S.template get<k>();
169     }

```

```

169 }
170 } // 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.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file SimplexSet.h
27  * @brief SimplexSet data structure and associated convenience functions.
28  */
29
30 #pragma once
31
32 #include <algorithm>
33 #include <unordered_set>
34 #include "util.h"
35
36 namespace casc
37 {
38
39 /**
40  * @brief A multiset to store simplices in a simplicial_complex.
41  *
42  * This is really a tuple of sets where each set corresponds to a simplex
43  * dimension. Many convenience functions are wrapped so this behaves much like
44  * a std::set.
45  *
46  * @tparam Complex Typename of the simplicial_complex.
47  */
48 template <typename Complex>
49 struct SimplexSet
50 {
51     /// Alias for SimplexID
52     template <std::size_t j>
53     using SimplexID = typename Complex::template SimplexID<j>;
54     /// Index sequence of types from the simplicial_complex
55     using LevelIndex = typename Complex::LevelIndex;
56     /// Index sequence starting at 1
57     using cLevelIndex = typename util::remove_first_val<std::size_t,
58                                                         LevelIndex>::type;
59     /// Reversed index sequence
60     using RevIndex = typename util::reverse_sequence<std::size_t,
61                                                         LevelIndex>::type;
62     /// Reversed index sequence stops at 1
63     using cRevIndex = typename util::reverse_sequence<std::size_t,
64                                                         cLevelIndex>::type;
65     /// Typename of this
66     using type_this = SimplexSet<Complex>;
67
68     /// Tuple of SimplexIDs wrt an integral level.
69     using SimplexIDLevel = typename util::int_type_map<std::size_t,
70                                                         std::tuple, LevelIndex, SimplexID>::type;
71     // No real sense to hide this tuple of sets from the end users.
72     // Making it private, we'd have to introduce lots of friend structs.
73     /// Tuple of NodeSets per level.
74     typename util::type_map<SimplexIDLevel, NodeSet>::type tupleSet;
75
76     /// Default constructor
77     SimplexSet() {}
78     /// Default destructor
79     ~SimplexSet() {}
80
81     // type_this& operator=(const type_this& other){
82     //     util::int_for_each<std::size_t, LevelIndex>(CopyHelper(), this, other);

```

```

83 // }
84
85 // type_this& operator=(type_this&& other){
86 //     util::int_for_each<std::size_t, LevelIndex>(CopyHelper(), this, other);
87 // }
88
89 /**
90  * @brief      Checks if a level has no elements.
91  *
92  * @tparam      k      Level to check.
93  *
94  * @return      True if the container is empty, false otherwise.
95  */
96 template <std::size_t k>
97 inline auto empty() const noexcept{
98     return std::get<k>(tupleSet).empty();
99 }
100
101 /**
102  * @brief      Return the number of elements in a level.
103  *
104  * @tparam      k      Simplex dimension to query
105  *
106  * @return      Returns the number of simplices of dimension 'k' are in the
107  *              set.
108  */
109 template <std::size_t k>
110 inline auto size() const noexcept{
111     return std::get<k>(tupleSet).size();
112 }
113
114 /**
115  * @brief      Clear the contents.
116  */
117 void clear()
118 {
119     util::int_for_each<std::size_t, LevelIndex>(ClearHelper(), this);
120 }
121
122 /**
123  * @brief      Insert a simplex into the set.
124  *
125  * @param[in]   s      Simplex to insert.
126  *
127  * @tparam      k      Simplex dimension of 's'.
128  */
129 template <std::size_t k>
130 inline void insert(SimplexID<k> s)
131 {
132     std::get<k>(tupleSet).insert(s);
133 }
134
135 /**
136  * @brief      Insert a SimplexSet into this.
137  *
138  * @param[in]   s      The SimplexSet to insert.
139  */
140 void insert(const SimplexSet<Complex> &s)
141 {
142     util::int_for_each<std::size_t, LevelIndex>(
143         InsertHelper(), this, s);
144 }
145
146 /**
147  * @brief      Remove a simplex from the set.
148  *
149  * @param[in]   s      Simplex to remove.
150  *
151  * @tparam      k      Simplex dimension of 's'.
152  */
153 template <std::size_t k>
154 inline void erase(SimplexID<k> s)
155 {
156     std::get<k>(tupleSet).erase(s);
157 }
158
159 /**
160  * @brief      Remove a set of simplices.
161  *
162  * @param[in]   s      SimplexSet to remove.
163  */
164 void erase(const SimplexSet<Complex> &s)
165 {
166     util::int_for_each<std::size_t, LevelIndex>(
167         EraseHelper(), this, s);
168 }
169

```



```

170  /**
171   * @brief      Get the simplex of interest.
172   *
173   * @param[in]  s      The simplex to search for.
174   *
175   * @tparam     k      Simplex dimension of 's'.
176   *
177   * @return     Iterator to an element with key equivalent to s. If no such
178   *             element is found, past-the-end iterator (see end()) is
179   *             returned.
180   */
181  template <std::size_t k>
182  inline auto find(const SimplexID<k> s)
183  {
184      return std::get<k>(tupleSet).find(s);
185  }
186
187  /**
188   * @brief      Get the simplex of interest.
189   *
190   * @param[in]  s      The simplex to search for.
191   *
192   * @tparam     k      Simplex dimension of 's'.
193   *
194   * @return     Iterator to an element with key equivalent to s. If no such
195   *             element is found, past-the-end iterator (see end()) is
196   *             returned.
197   */
198  template <std::size_t k>
199  inline auto find(const SimplexID<k> s) const
200  {
201      return std::get<k>(tupleSet).find(s);
202  }
203
204  /**
205   * @brief      Get the past-the-end iterator.
206   *
207   * @tparam     k      The simplex dimension to get iterator of.
208   *
209   * @return     Returns an iterator to the element following the last element
210   *             of the set for the specified simplex dimension.
211   */
212  template <std::size_t k>
213  inline auto end()
214  {
215      return std::get<k>(tupleSet).end();
216  }
217
218  /**
219   * @brief      Get the past-the-end iterator.
220   *
221   * @tparam     k      The simplex dimension to get iterator of.
222   *
223   * @return     Returns an iterator to the element following the last element
224   *             of the set for the specified simplex dimension.
225   */
226  template <std::size_t k>
227  inline auto cend() const
228  {
229      return std::get<k>(tupleSet).cend();
230  }
231
232  /**
233   * @brief      Get an iterator to the first element of the container.
234   *
235   * @tparam     k      The simplex dimension to get iterator of.
236   *
237   * @return     Returns an iterator to the first element.
238   */
239  template <std::size_t k>
240  inline auto begin()
241  {
242      return std::get<k>(tupleSet).begin();
243  }
244
245  /**
246   * @brief      Get an iterator to the first element of the container.
247   *
248   * @tparam     k      The simplex dimension to get iterator of.
249   *
250   * @return     Returns an iterator to the first element.
251   */
252  template <std::size_t k>
253  inline auto cbegin() const
254  {
255      return std::get<k>(tupleSet).cbegin();
256  }

```

```

257
258 // /**
259 // * @brief      Get the NodeSet for a particular simplex dimension.
260 // *
261 // * @tparam      k      Simplex dimension to get.
262 // *
263 // * @return      Returns the NodeSet corresponding to the requested dimension.
264 // */
265 template <std::size_t k>
266 inline auto &get()
267 {
268     return std::get<k>(tupleSet);
269 }
270
271 // /**
272 // * @brief      Get the NodeSet for a particular simplex dimension.
273 // *
274 // * @tparam      k      Simplex dimension to get.
275 // *
276 // * @return      Returns the NodeSet corresponding to the requested dimension.
277 // */
278 template <std::size_t k>
279 inline auto &get() const
280 {
281     return std::get<k>(tupleSet);
282 }
283
284 /**
285 * @brief      Print the SimplexSet.
286 *
287 * See also casc::simplicial_complex::SimplexID::operator«.
288 *
289 * @param      output      Handle to the stream to print to.
290 * @param[in]   S           SimplexSet to print.
291 *
292 * @return      Handle to the stream.
293 */
294 friend std::ostream &operator<<(std::ostream &output, const SimplexSet<Complex> &S)
295 {
296     output << "SimplexSet(";
297     util::int_for_each<std::size_t, LevelIndex>(PrintHelper(),
298                                                output, S);
299     output << ")";
300     return output;
301 }
302
303
304 private:
305     /**
306     * @brief      Helper struct to insert a SimplexSet.
307     */
308     struct InsertHelper
309     {
310         /**
311         * @brief      Perform the insertion for a dimension.
312         *
313         * @param      that      Typename of this SimplexSet.
314         * @param[in]   S         SimplexSet to insert
315         *
316         * @tparam      k         Simplex dimension to insert.
317         */
318         template <std::size_t k>
319         static void apply(type_this* that, const SimplexSet<Complex> &S)
320         {
321             auto s = std::get<k>(S.tupleSet);
322             for (auto simplex : s)
323             {
324                 that->insert(simplex);
325             }
326         }
327     };
328
329     /**
330     * @brief      Helper struct to compute a set difference.
331     */
332     struct EraseHelper
333     {
334         /**
335         * @brief      Perform the set difference for a dimension.
336         *
337         * @param      that      Typename of this SimplexSet.
338         * @param[in]   S         SimplexSet to remove from this.
339         *
340         * @tparam      k         Simplex dimension to erase.
341         */
342         template <std::size_t k>
343         static void apply(type_this* that, const SimplexSet<Complex> &S)

```

```

344         {
345             auto s = std::get<k>(S.tupleSet);
346             for (auto simplex : s)
347             {
348                 that->erase(simplex);
349             }
350         }
351     };
352
353     /**
354      * @brief      Helper struct to print the SimplexSet
355      */
356     struct PrintHelper
357     {
358         /**
359          * @brief      Print the simplices in the level.
360          *
361          * @param      output    Handle to the stream to output to.
362          * @param[in]  S          SimplexSet to print.
363          *
364          * @tparam      k          Simplex dimension to print.
365          */
366         template <std::size_t k>
367         static void apply(std::ostream &output, const SimplexSet<Complex> &S)
368         {
369             output << "[l=" << k;
370             auto s = std::get<k>(S.tupleSet);
371             for (auto simplex : s)
372             {
373                 output << ", " << simplex;
374             }
375             output << "];";
376         }
377     };
378
379     /**
380      * @brief      Helper struct to clear the SimplexSet.
381      */
382     struct ClearHelper
383     {
384         /**
385          * @brief      Clear a dimension.
386          *
387          * @param      that      Typename of this SimplexSet.
388          *
389          * @tparam      k          Simplex dimension to clear.
390          */
391         template <std::size_t k>
392         void apply(type_this* that)
393         {
394             auto &s = std::get<k>(that->tupleSet);
395             s.clear();
396         }
397     };
398
399     // struct CopyHelper
400     // {
401     //     template <std::size_t k>
402     //     void apply(type_this& that, type_this& other){
403     //         auto &s = that.get<k>();
404     //         s = other.get<k>();
405     //     }
406     //
407     //     template <std::size_t k>
408     //     void apply(type_this& that, type_this&& other){
409     //         auto &s = that.get<k>();
410     //         s = other.get<k>();
411     //     }
412     // };
413 };
414
415 /**
416 * @brief      Get the NodeSet for a simplex dimension from a SimplexSet.
417 *
418 * @param      S          SimplexSet of interest.
419 *
420 * @tparam      k          Simplex dimension desired.
421 * @tparam      Complex    Typename of the simplicial_complex.
422 *
423 * @return     A NodeSet which holds simplices of dimension 'k' and a member of
424 *             SimplexSet 'S'.
425 */
426 template <std::size_t k, typename Complex>
427 static inline auto &get(SimplexSet<Complex> &S)
428 {
429     return S.template get<k>();
430 }

```

```

431
432 /**
433  * @overload
434  */
435 template <std::size_t k, typename Complex>
436 static inline auto &get(const SimplexSet<Complex> &S)
437 {
438     return S.template get<k>();
439 }
440
441 /// @cond detail
442 /// Namespace for simplex container related helpers
443 namespace simplex_set_detail
444 {
445
446 /**
447  * @brief      Helper struct to compute the union of two SimplexSets.
448  *
449  * @tparam      Complex  Typename of the simplicial_complex.
450  */
451 template <typename Complex>
452 struct UnionH
453 {
454     /**
455      * @brief      Compute the union of two SimplexSets.
456      *
457      * \f$A \cup B \f$
458      *
459      * @param[in]  A      A SimplexSet
460      * @param[in]  B      Another SimplexSet
461      * @param[out] dest   The destination SimplexSet
462      *
463      * @tparam      k      The current simplex dimension to merge.
464      */
465     template <std::size_t k>
466     static void apply(const SimplexSet<Complex> &A,
467                     const SimplexSet<Complex> &B,
468                     SimplexSet<Complex> &dest)
469     {
470         auto a = std::get<k>(A.tupleSet);
471         auto b = std::get<k>(B.tupleSet);
472         auto &d = std::get<k>(dest.tupleSet);
473         d.insert(a.begin(), a.end());
474         d.insert(b.begin(), b.end());
475     }
476 };
477
478 /**
479  * @brief      Helper struct to compute the intersection of two SimplexSets.
480  *
481  * @tparam      Complex  Typename of the simplicial_complex.
482  */
483 template <typename Complex>
484 struct IntersectH
485 {
486     /**
487      * @brief      Compute the intersection of two SimplexSets.
488      *
489      * \f$A \cap B \f$
490      *
491      * @param[in]  A      A SimplexSet
492      * @param[in]  B      Another SimplexSet
493      * @param      dest   The destination SimplexSet.
494      *
495      * @tparam      k      The current simplex dimension to merge.
496      */
497     template <std::size_t k>
498     static void apply(const SimplexSet<Complex> &A,
499                     const SimplexSet<Complex> &B,
500                     SimplexSet<Complex> &dest)
501     {
502         auto a = casc::get<k>(A);
503         auto b = casc::get<k>(B);
504         auto &d = casc::get<k>(dest);
505
506         if (a.size() < b.size())
507         {
508             for (auto item : a)
509             {
510                 if (b.find(item) != b.end())
511                     d.insert(item);
512             }
513         }
514         else
515         {
516             for (auto item : b)
517             {

```

```

518             if (a.find(item) != a.end())
519                 d.insert(item);
520         }
521     }
522 }
523 };
524
525 /**
526  * @brief      Helper struct to compute the set intersection.
527  * @tparam      Complex  Typename of the simplicial_complex.
528  */
529 template <typename Complex>
530 struct DifferenceH
531 {
532     /**
533      * @brief      Compute the set difference for a simplex dimension.
534      *
535      * \f$ dest = A \setminus B \f$
536      *
537      * @param[in]  A      A SimplexSet.
538      * @param[in]  B      Remove this SimplexSet from A.
539      * @param      dest    The destination SimplexSet.
540      *
541      * @tparam      k      The simplex dimension to compute the difference of.
542      */
543     template<std::size_t k>
544     static void apply(const SimplexSet<Complex> &A,
545                     const SimplexSet<Complex> &B,
546                     SimplexSet<Complex> &dest)
547     {
548         auto a = casc::get<k>(A);
549         auto b = casc::get<k>(B);
550         auto &d = casc::get<k>(dest);
551
552         for (auto item : a)
553         {
554             if (b.find(item) == b.end())
555                 d.insert(item);
556         }
557     }
558 };
559 };
560
561 /**
562  * @brief      Helper struct to compute set equivalence.
563  *
564  * @tparam      Complex  Typename of the simplicial_complex.
565  */
566 template <typename Complex>
567 struct OperatorEQH
568 {
569     /// Result of the comparison
570     bool result;
571
572     /// Default constructor
573     OperatorEQH(): result(true) {}
574
575     /**
576      * @brief      Compare the two sets by level.
577      *
578      * @param[in]  lhs    The left hand side
579      * @param[in]  rhs    The right hand side
580      *
581      * @tparam      k      Level to compare.
582      */
583     template <std::size_t k>
584     void apply(const SimplexSet<Complex> &lhs,
585              const SimplexSet<Complex> &rhs) {
586         auto a = casc::get<k>(lhs);
587         auto b = casc::get<k>(rhs);
588         result &= a==b;
589     }
590 };
591 } // end namespace simplex_set_detail
592 /// @endcond
593
594 /**
595  * @brief      Compare if the sets are equivalent
596  *
597  * @param[in]  lhs    The left hand side
598  * @param[in]  rhs    The right hand side
599  *
600  * @tparam      Complex  Typename of the simplicial_complex
601  *
602  * @return      True if the sets are equal, false otherwise.
603  */
604 template <typename Complex>

```

```

605 bool operator==(const SimplexSet<Complex> &lhs, const SimplexSet<Complex> &rhs){
606     auto func = simplex_set_detail::OperatorEQH<Complex>();
607     util::int_for_each<std::size_t, typename Complex::LevelIndex>(
608         func, lhs, rhs);
609     return func.result;
610 }
611
612 /**
613  * @brief      Compare if the sets are not equivalent.
614  * @param[in]  lhs      The left hand side
615  * @param[in]  rhs      The right hand side
616  * @tparam     Complex  Typename of the simplicial_complex.
617  * @return     True if the sets are unequal, false otherwise.
618  */
619 template <typename Complex>
620 bool operator!=(const SimplexSet<Complex> &lhs, const SimplexSet<Complex> &rhs){
621     return !(lhs == rhs);
622 }
623
624 /**
625  * @brief      Compute the set union.
626  * @param[in]  A          A SimplexSet
627  * @param[in]  B          Another SimplexSet
628  * @param[out] dest       The destination SimplexSet.
629  * @tparam     Complex    Typename of the simplicial_complex.
630  */
631 template <typename Complex>
632 static void set_union(const SimplexSet<Complex> &A,
633                     const SimplexSet<Complex> &B,
634                     SimplexSet<Complex> &dest)
635 {
636     util::int_for_each<std::size_t,
637         typename Complex::LevelIndex>(
638         simplex_set_detail::UnionH<Complex>(), A, B, dest);
639 }
640
641 /**
642  * @brief      Compute the set intersection.
643  * @param[in]  A          A SimplexSet
644  * @param[in]  B          Another SimplexSet
645  * @param[out] dest       The destination SimplexSet.
646  * @tparam     Complex    Typename of the simplicial_complex.
647  */
648 template <typename Complex>
649 static void set_intersection(const SimplexSet<Complex> &A,
650                             const SimplexSet<Complex> &B,
651                             SimplexSet<Complex> &dest)
652 {
653     util::int_for_each<std::size_t,
654         typename Complex::LevelIndex>(
655         simplex_set_detail::IntersectH<Complex>(), A, B, dest);
656 }
657
658 /**
659  * @brief      Compute the set difference.
660  * @param[in]  A          A SimplexSet
661  * @param[in]  B          Another SimplexSet
662  * @param[out] dest       The destination SimplexSet.
663  * @tparam     Complex    Typename of the simplicial_complex.
664  */
665 template <typename Complex>
666 static void set_difference(const SimplexSet<Complex> &A,
667                           const SimplexSet<Complex> &B,
668                           SimplexSet<Complex> &dest)
669 {
670     util::int_for_each<std::size_t,
671         typename Complex::LevelIndex>(
672         simplex_set_detail::DifferenceH<Complex>(), A, B, dest);
673 }
674
675 // 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 <stdint>
#include <map>
#include <set>
#include <iterator>
#include <array>
#include <vector>
#include <iostream>
#include <fstream>
#include <functional>
#include <type_traits>
#include <ostream>
#include <unordered_set>
#include <unordered_map>
#include <utility>
#include <stdexcept>
#include "index_tracker.h"
#include "util.h"

```

Data Structures

- class [casc::simplicial_complex< traits >](#)
The CASC data structure for representing simplicial complexes of arbitrary dimensionality with coloring.
- struct [casc::simplicial_complex< traits >::SimplexID< k >](#)
A handle for a simplex object in the complex.
- struct [casc::simplicial_complex< traits >::EdgeID< k >](#)
External reference to an edge or a connection within the complex.

Namespaces

- namespace [casc](#)
Namespace for everything CASC.

Typedefs

- template<typename KeyType, typename ... Ts>
using [casc::AbstractSimplicialComplex](#) = simplicial_complex< detail::simplicial_complex_traits_default< KeyType, Ts... > >
- template<typename T>
using [casc::NodeSet](#) = std::unordered_set< T, simplex_set_detail::hashSimplexID< T > >
Helpful alias defining an unordered_set of simplices. See also hashSimplexID.

10.16 SimplicialComplex.h

[Go to the documentation of this file.](#)

```

1  /*
2  * ****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  *   and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either
11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * ****
23 */
24
25 /**
26 * @file   SimplicialComplex.h
27 * @brief  This header contains the main CASC data structure and associated
28 *         components.
29 */
30
31 #pragma once
32
33 #include <algorithm>
34 #include <assert.h>
35 #include <stdint_t>
36 #include <map>
37 #include <set>
38 #include <iterator>
39 #include <array>
40 #include <vector>
41 #include <iostream>
42 #include <fstream>
43 #include <functional>
44 #include <type_traits>
45 #include <ostream>
46 #include <unordered_set>
47 #include <unordered_map>
48 #include <utility>
49 #include <stdexcept>
50
51 #include "index_tracker.h"
52 #include "util.h"
53
54 #if __has_cpp_attribute(maybe_unused)
55 #define MAYBE_UNUSED [[maybe_unused]]
56 #else
57 #define MAYBE_UNUSED
58 #endif
59
60 /// Namespace for everything CASC
61 namespace casc
62 {
63     /// @cond detail
64     /// Namespace for CASC internal data structures
65     namespace detail
66     {
67         /// Data structure to store simplices by level.
68         template <class T> using map = std::map<std::size_t, T>;
69
70         /**
71          * @brief      A generic pair type representing Key to Value associations.
72          *
73          * @tparam      T1      Typename of the Key
74          * @tparam      T2      Typename of the Value
75          */
76         template <typename T1, typename T2>
77         struct asc_pair {
78             using this_t = asc_pair<T1, T2>;
79             asc_pair() {}
80             asc_pair(const T1& first, const T2& second) : _pair(first, second) {}
81             asc_pair(T1&& first, T2&& second) : _pair(std::forward<T1>(first), std::forward<T2>(second)) {}
82             asc_pair(const this_t& other) : _pair(other._pair) {}

```



```

83     asc_pair(this_t&& other) : _pair(std::forward<std::pair<T1,T2>(other._pair)) {}
84
85     operator T1() const {
86         return _pair.first;
87     }
88
89     this_t& operator=(const this_t& other){
90         _pair = other._pair;
91         return *this;
92     }
93
94     this_t& operator=(this_t&& other){
95         _pair = std::move(other._pair);
96         return *this;
97     }
98
99     friend bool operator==(const this_t& lhs, const this_t& rhs) {return lhs.first == rhs.first;}
100    friend bool operator!=(const this_t& lhs, const this_t& rhs) {return lhs.first != rhs.first;}
101    friend bool operator<=(const this_t& lhs, const this_t& rhs) {return lhs.first <= rhs.first;}
102    friend bool operator>=(const this_t& lhs, const this_t& rhs) {return lhs.first >= rhs.first;}
103    friend bool operator<(const this_t& lhs, const this_t& rhs) {return lhs.first < rhs.first;}
104    friend bool operator>(const this_t& lhs, const this_t& rhs) {return lhs.first > rhs.first;}
105
106    T1& first = _pair.first;
107    T2& second = _pair.second;
108 private:
109    std::pair<T1,T2> _pair;
110 };
111
112 /**
113  * @brief      Array of asc_pairs sorted by Key for boundary adjacency storage.
114  *
115  * @tparam     KEY_T  Typename of Key
116  * @tparam     VAL_T  Typename of Value
117  * @tparam     k      Size of the array
118  */
119 template <typename KEY_T, typename VAL_T, std::size_t k>
120 struct asc_arraymap {
121     using pair_t = asc_pair<KEY_T, VAL_T>;
122     using array_t = std::array<pair_t, k>;
123     using iterator = typename array_t::iterator;
124     using const_iterator = typename array_t::const_iterator;
125
126     asc_arraymap(){
127         _begin = _array.begin();
128         _end = _array.begin();
129     }
130
131     void insert(pair_t& p){
132         if (_end == _array.end())
133             throw std::out_of_range("insert&: Adding element beyond the end of array.");
134         *_end = p;
135         ++_end;
136         std::sort(_begin, _end);
137     }
138
139     void insert(pair_t&& p){
140         if (_end == _array.end())
141             throw std::out_of_range("insert&&: Adding element beyond the end of array.");
142         *_end = std::forward<pair_t>(p);
143         ++_end;
144         std::sort(_begin, _end);
145     }
146
147     iterator find(const KEY_T& key){
148         return std::find(_begin, _end, key);
149     }
150
151     void erase(const KEY_T& key){
152         auto it = std::find(_begin, _end, key);
153         if(it != _end){
154             std::copy(it+1, _end, it);
155             --_end;
156         }
157     }
158
159     std::size_t size() const{
160         return std::distance(_end, _begin);
161     }
162
163     VAL_T& operator[](const KEY_T& key){
164         auto it = std::find(_begin, _end, key);
165         if(it != _end){
166             return it->second;
167         }
168         else{
169             if (_end == _array.end())

```

```

170         throw std::out_of_range("operator[]: Adding element beyond the end of array.");
171         _end->first = key;
172         ++_end;
173         std::sort(_begin, _end);
174         return std::find(_begin, _end, key)->second;
175     }
176 }
177
178 iterator begin(){ return _begin; }
179 iterator end(){ return _end; }
180 const_iterator cbegin() const {return _begin;}
181 const_iterator cend() const {return _end;}
182
183 private:
184     array_t _array;
185     iterator _begin;
186     iterator _end;
187 };
188
189
190 /**
191  * @brief      Sorted vector of asc_pairs for coboundary relation storage.
192  *
193  * @tparam     KEY_T  Typename of Key
194  * @tparam     VAL_T  Typename of Values
195  */
196 template <typename KEY_T, typename VAL_T>
197 struct asc_vectormap {
198     using pair_t = asc_pair<KEY_T, VAL_T>;
199     using vector_t = std::vector<pair_t>;
200     using iterator = typename vector_t::iterator;
201     using const_iterator = typename vector_t::const_iterator;
202
203     asc_vectormap() {}
204
205     void insert(pair_t& p){
206         iterator first = std::lower_bound(_vector.begin(), _vector.end(), p);
207         if ((first == _vector.end()) || (*first != p)){
208             _vector.insert(first, p);
209         }
210         else{
211             std::cout << "Item already exists...";
212         }
213     }
214
215     void insert(pair_t&& p){
216         iterator first = std::lower_bound(_vector.begin(), _vector.end(), p);
217         if ((first == _vector.end()) || (*first != p)){
218             _vector.insert(first, std::forward<pair_t>(p));
219         }
220         else{
221             std::cout << "Item already exists...";
222         }
223     }
224
225     iterator find(const KEY_T& key){
226         iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
227         if (first != _vector.end()){
228             if (*first != key){
229                 return _vector.end();
230             }
231             else{
232                 return first;
233             }
234         }
235         else{
236             return first;
237         }
238     }
239
240     void erase(const KEY_T& key){
241         iterator it = this->find(key);
242         if (it != _vector.end()){
243             _vector.erase(it);
244         }
245     }
246
247     std::size_t size() const{
248         return _vector.size();
249     }
250
251     VAL_T& at(const KEY_T& key){
252         iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
253         if ((first == _vector.end()) || (first->first != key)){
254             throw std::out_of_range("Could not find element in asc_vectormap.");
255         }
256         else {

```

```

257         return first->second;
258     }
259 }
260
261 VAL_T& operator[] (const KEY_T& key) {
262     iterator first = std::lower_bound(_vector.begin(), _vector.end(), key);
263     if ((first == _vector.end()) || (first->first != key)) {
264         first = _vector.emplace(first, pair_t());
265         first->first = key;
266         return first->second;
267     }
268     else {
269         return first->second;
270     }
271 }
272
273 iterator begin() { return _vector.begin(); }
274 iterator end() { return _vector.end(); }
275 const_iterator cbegin() const { return _vector.cbegin(); }
276 const_iterator cend() const { return _vector.cend(); }
277 private:
278     vector_t _vector;
279 };
280
281
282 /**
283  * @brief Template prototype for Nodes in CASC.
284  *
285  * asc_Node must be defined outside of simplicial_complex because C++ does
286  * not allow internal templates to be partially specialized. This template
287  * prototype is later specialized to represent various Node roles.
288  */
289 template <class KeyType, std::size_t k, std::size_t N, typename DataTypes, class> struct asc_Node;
290
291 /// This is the base Node class.
292 struct asc_NodeBase {
293     /**
294      * @brief Construct a Node
295      *
296      * @param[in] id An internal integer identifier of the Node.
297      */
298     asc_NodeBase(std::size_t id) : _node(id) {}
299     virtual ~asc_NodeBase() {}; /**< Destructor */
300     std::size_t _node; /**< Internal Node ID*/
301 };
302
303 /**
304  * @brief Base class for Node with some data.
305  *
306  * @tparam DataType Typename of the data to be stored.
307  */
308 template <class DataType>
309 struct asc_NodeData {
310     DataType _data; /**< stored data with type DataType */
311 };
312
313 /**
314  * @brief Explicit specialization for Nodes without data.
315  *
316  * This exists so that the compiler knows to not allocate any memory to
317  * store data when void is specified.
318  */
319 template <>
320 struct asc_NodeData<void> {};
321
322 /**
323  * @brief Base class for Nodes with edge data.
324  *
325  * @tparam KeyType Typename of index for indexing Nodes.
326  * @tparam DataType Typename of the data stored on the edge.
327  */
328 template <class KeyType, class DataType>
329 struct asc_EdgeData {
330     /** The map of SimplexIDs to stored edge data. */
331     std::unordered_map<KeyType, DataType> _edge_data;
332 };
333
334 /**
335  * @brief Explicit specialization for Nodes with no edge data.
336  *
337  * @tparam KeyType Typename of index for indexing Nodes.
338  */
339 template <class KeyType>
340 struct asc_EdgeData<KeyType, void> {};
341
342 /**
343  * @brief Base class for Node with parent nodes

```

```

344 *
345 * @tparam KeyType      Typename of the Node index
346 * @tparam k            The Simplex dimension
347 * @tparam N            Dimension of the Complex
348 * @tparam NodeDataTypes A util::type_holder array of Node types
349 * @tparam EdgeDataTypes A util::type_holder array of Edge types
350 */
351 template < class KeyType,
352            std::size_t k,
353            std::size_t N,
354            class NodeDataTypes,
355            class EdgeDataTypes>
356 struct asc_NodeDown :
357     public asc_EdgeData<KeyType,
358         typename util::type_get<k-1, EdgeDataTypes>::type> {
359     /** Alias the typename of the parent Node */
360     using DownNodeT = asc_Node<KeyType, k-1, N, NodeDataTypes, EdgeDataTypes>;
361
362     /** Map of indices to parent Node pointers*/
363     asc_arraymap<KeyType, DownNodeT*, k> _down;
364     // std::map<KeyType, DownNodeT*> _down;
365 };
366
367 /**
368 * @brief      Base class for Node with children Nodes
369 *
370 * @tparam KeyType      Typename of the Node index
371 * @tparam k            The Simplex dimension
372 * @tparam N            Dimension of the Complex
373 * @tparam NodeDataTypes A util::type_holder array of Node types
374 * @tparam EdgeDataTypes A util::type_holder array of Edge types
375 */
376 template < class KeyType,
377            std::size_t k,
378            std::size_t N,
379            class NodeDataTypes,
380            class EdgeDataTypes>
381 struct asc_NodeUp {
382     /// Typename of the nodes up.
383     using UpNodeT = asc_Node<KeyType, k+1, N, NodeDataTypes, EdgeDataTypes>;
384     asc_vectormap<KeyType, UpNodeT*> _up;
385     // std::unordered_map<KeyType, UpNodeT*> _up;      /**< @brief Map of pointers to children */
386 };
387
388 /**
389 * @brief      Node with both parents and children
390 *
391 * @tparam KeyType      Typename of the Node index
392 * @tparam k            The Simplex dimension
393 * @tparam N            Dimension of the Complex
394 * @tparam NodeDataTypes A util::type_holder of Node types
395 * @tparam EdgeDataTypes A util::type_holder of Edge types
396 */
397 template <class KeyType, std::size_t k, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
398 struct asc_Node : public asc_NodeBase,
399     public asc_NodeData<typename util::type_get<k, NodeDataTypes>::type>,
400     public asc_NodeDown<KeyType, k, N, NodeDataTypes, EdgeDataTypes>,
401     public asc_NodeUp<KeyType, k, N, NodeDataTypes, EdgeDataTypes>
402 {
403     /// Dimension of the simplex.
404     static constexpr std::size_t level = k;
405
406     /**
407     * @brief      Default constructor
408     *
409     * @param[in] id    The internal integer identifier.
410     */
411     asc_Node(std::size_t id) : asc_NodeBase(id) {}
412
413     /**
414     * @brief      Print the Node out for debugging only
415     *
416     * @param      output    The output stream.
417     * @param[in] node    The Node of interest to print.
418     *
419     * @return     A handle to the output stream.
420     */
421     friend std::ostream &operator<<(std::ostream &output, const asc_Node &node)
422     {
423         output << "Node(level=" << k << ", " << "id=" << node._node;
424         if (node._down.size() > 0)
425         {
426             for (auto it = node._down.cbegin(); it != node._down.cend(); ++it)
427             {
428                 output << ", NodeDownID={'"
429                     << it->first << ", "
430                     << it->second->_node << "}";

```

```

431     }
432 }
433 if (node._up.size() > 0)
434 {
435     for (auto it = node._up.cbegin(); it != node._up.cend(); ++it)
436     {
437         output << " , NodeUpID={'"
438             << it->first << " , "
439             << it->second->_node << " }";
440     }
441 }
442 output << " )";
443 return output;
444 }
445 };
446
447 /**
448  * @brief      Node with only children i.e., the root.
449  *
450  * @tparam      KeyType          Typename of the Node index
451  * @tparam      N                The Simplex dimension
452  * @tparam      NodeDataTypes    A util::type_holder of Node types
453  * @tparam      EdgeDataTypes    A util::type_holder of Edge types
454  */
455 template <class KeyType, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
456 struct asc_Node<KeyType, 0, N, NodeDataTypes, EdgeDataTypes> :
457     public asc_NodeBase,
458     public asc_NodeData<typename util::type_get<0, NodeDataTypes>::type>,
459     public asc_NodeUp<KeyType, 0, N, NodeDataTypes, EdgeDataTypes>
460 {
461     /// Dimension of the simplex.
462     static constexpr std::size_t level = 0;
463
464     /**
465      * @brief      Default constructor
466      *
467      * @param[in]   id        The internal integer identifier.
468      */
469     asc_Node(std::size_t id) : asc_NodeBase(id) {}
470
471     /**
472      * @brief      Print the Node out for debugging only
473      *
474      * @param       output    The output stream.
475      * @param[in]   node      The Node of interest to print.
476      *
477      * @return      A handle to the output stream.
478      */
479     friend std::ostream &operator<<(std::ostream &output, const asc_Node &node)
480     {
481         output << "Node(level=" << 0
482             << " , id=" << node._node;
483         if (node._up.size() > 0)
484         {
485             for (auto it = node._up.cbegin(); it != node._up.cend(); ++it)
486             {
487                 output << " , NodeUpID={'"
488                     << it->first << " , "
489                     << it->second->_node << " }";
490             }
491         }
492         output << " )";
493         return output;
494     }
495 };
496
497 /**
498  * @brief      Top level node with only parents
499  *
500  * @tparam      KeyType          Typename of the Node index
501  * @tparam      N                The Simplex dimension
502  * @tparam      NodeDataTypes    A util::type_holder of Node types
503  * @tparam      EdgeDataTypes    A util::type_holder of Edge types
504  */
505 template <class KeyType, std::size_t N, class NodeDataTypes, class EdgeDataTypes>
506 struct asc_Node<KeyType, N, N, NodeDataTypes, EdgeDataTypes> :
507     public asc_NodeBase,
508     public asc_NodeData<typename util::type_get<N, NodeDataTypes>::type>,
509     public asc_NodeDown<KeyType, N, N, NodeDataTypes, EdgeDataTypes>
510 {
511     /// Dimension of the simplex.
512     static constexpr std::size_t level = N;
513
514     /**
515      * @brief      Default constructor
516      *
517      * @param[in]   id        The internal integer identifier.

```

```

518     */
519     asc_Node(std::size_t id) : asc_NodeBase(id) {}
520
521     /**
522     * @brief      Print the Node out for debugging only
523     *
524     * @param      output  The output stream.
525     * @param[in]   node    The Node of interest to print.
526     *
527     * @return      A handle to the output stream.
528     */
529     friend std::ostream &operator<<(std::ostream &output, const asc_Node &node)
530     {
531         output << "Node(level=" << N
532             << ", id=" << node._node;
533         if (node._down.size() > 0)
534         {
535             for (auto it = node._down.cbegin(); it != node._down.cend(); ++it)
536             {
537                 output << ", NodeDownID={'"
538                     << it->first << ", "
539                     << it->second->_node << "}";
540             }
541         }
542         output << ")";
543         return output;
544     }
545 };
546
547 /**
548 * @brief      An iterator adapter to iterate over NodeIDs.
549 *
550 * @tparam      Iter  Typename of the iterator
551 * @tparam      Data  Typename of the data
552 */
553 template <typename Iter, typename Data>
554 struct node_id_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
555     public:
556         /// Inherit from a bidirectional std::iterator.
557         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
558         /// Empty constructor
559         node_id_iterator() {}
560         /// Instantiate with an iterator to wrap
561         node_id_iterator(Iter j) : i(j) {}
562         /// Increment the iterator
563         node_id_iterator &operator++() { ++i; return *this; }
564         /// Increment the iterator
565         node_id_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
566         /// Decrement the iterator
567         node_id_iterator &operator--() { --i; return *this; }
568         /// Decrement the iterator
569         node_id_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
570         /// Iterator equality comparison
571         bool operator==(const node_id_iterator &j) const { return i == j.i; }
572         /// Iterator inequality comparison
573         bool operator!=(const node_id_iterator &j) const { return !(*this == j); }
574         /// Dereferencing the iterator produces a SimplexID.
575         Data operator*() const { return Data(i->second); }
576         /// Const version
577         const Data operator*() const { return Data(i->second); }
578         /// Dereferencing the iterator produces a SimplexID.
579         typename super::pointer operator->() const { return Data(i->second); }
580     protected:
581         /// The iterator to wrap.
582         Iter i;
583 };
584
585 /**
586 * @brief      Convert an iterator into a node_id_iterator.
587 *
588 * @param[in]   j        The iterator to wrap
589 *
590 * @tparam      Iter  Typename of the iterator
591 * @tparam      Data  Typename of the data
592 *
593 * @return      An iterator over NodeIDs.
594 */
595 template <typename Iter, typename Data>
596 inline node_id_iterator<Iter, Data> make_node_id_iterator(Iter j)
597 {
598     return node_id_iterator<Iter, Data>(j);
599 }
600
601 /**
602 * @brief      An iterator adapter to iterate over Node data.
603 *
604 * @tparam      Iter  Typename of the iterator

```

```

605 * @tparam    Data    Typename of the data
606 */
607 template <typename Iter, typename Data>
608 struct node_data_iterator : public std::iterator<std::bidirectional_iterator_tag, Data> {
609     public:
610         /// Inherit from a bidirectional std::iterator.
611         using super = std::iterator<std::bidirectional_iterator_tag, Data>;
612         /// Empty constructor.
613         node_data_iterator() {}
614         /// Instantiate with an iterator to wrap.
615         node_data_iterator(Iter j) : i(j) {}
616         /// Increment the iterator
617         node_data_iterator &operator++() { ++i; return *this; }
618         /// Increment the iterator
619         node_data_iterator operator++(int) { auto tmp = *this; ++(*this); return tmp; }
620         /// Decrement the iterator
621         node_data_iterator &operator--() { --i; return *this; }
622         /// Decrement the iterator
623         node_data_iterator operator--(int) { auto tmp = *this; --(*this); return tmp; }
624         /// Iterator comparison
625         bool operator==(node_data_iterator j) const { return i == j.i; }
626         /// Iterator inequality comparison
627         bool operator!=(node_data_iterator j) const { return !(*this == j); }
628         /// Dereferencing the iterator produces the data.
629         typename super::reference operator*() { return i->second->_data; }
630         /// Dereferencing the iterator produces the data.
631         typename super::pointer operator->() { return i->second->_data; }
632     protected:
633         /// The wrapped iterator.
634         Iter i;
635 };
636
637 /**
638 * @brief      Convert an iterator into a node_data_iterator.
639 *
640 * @param[in]  j      The iterator to wrap
641 *
642 * @tparam     Iter    Typename of the iterator
643 * @tparam     Data    Typename of the data
644 *
645 * @return     An iterator over Node data.
646 */
647 template <typename Iter, typename Data>
648 inline node_data_iterator<Iter, Data> make_node_data_iterator(Iter j)
649 {
650     return node_data_iterator<Iter, Data>(j);
651 }
652
653 /**
654 * @brief      Helper to build a traits struct via expanding explicitly
655 * specified
656 * traits from AbstractSimplicialComplex.
657 *
658 * @tparam     K        Typename for the KeyType
659 * @tparam     Ts        Types of data to be stored on simplices.
660 */
661 template <typename K, typename ... Ts>
662 struct simplicial_complex_traits_default
663 {
664     /// Template to assign ints for all levels.
665     template <std::size_t k> using all_int = int;
666     /// Alias for KeyType
667     using KeyType = K;
668     /// The typenames of the data to be stored on simplices.
669     using NodeTypes = util::type_holder<Ts...>;
670     /// Assign all_int type to all edges
671     using EdgeTypes = typename util::int_type_map<std::size_t,
672                                                    util::type_holder,
673                                                    typename std::make_index_sequence<sizeof ... (Ts)-1>,
674                                                    all_int>::type;
675 };
676 } // end namespace detail
677 /// @endcond
678
679 /**
680 * @class      simplicial_complex
681 *
682 * @brief      The CASC data structure for representing simplicial complexes of
683 * arbitrary dimensionality with coloring.
684 *
685 * You can create a CASC object by defining a struct containing the
686 * traits of the complex. For example:
687 * ~~~~~{.cpp}
688 * struct complex_traits{
689 *     using KeyType = int;
690 *     using NodeTypes = util::type_holder<int,int,int,int>;
691 *     using EdgeTypes = util::type_holder<int,int,int>;

```

```

692 * };
693 *
694 * using SurfaceMesh = simplicial_complex<complex_traits>;
695 * ~~~~~
696 * This is the preferred method for creating a new CASC type. Alternatively you
697 * can use the ::AbstractSimplicialComplex alias to build a struct for you.
698 *
699 * @tparam      traits  A struct defining the dimension of the complex and data
700 *                      to be stored on each node and edge.
701 */
702 template <typename traits>
703 class simplicial_complex
704 {
705     public:
706         /// Typename of simplex keys.
707         using KeyType = typename traits::KeyType;
708         /// Typenames of the data stored on simplices.
709         using NodeDataTypes = typename traits::NodeTypes;
710         /// Typenames of the data stored on edges.
711         using EdgeDataTypes = typename traits::EdgeTypes;
712         /// Type of this
713         using type_this = simplicial_complex<traits>;
714         /// Total number of levels in the complex.
715         static constexpr std::size_t numLevels = NodeDataTypes::size;
716         /// Dimension of the simplicial complex.
717         static constexpr std::size_t topLevel = numLevels-1;
718         /// Dimension of boundaries.
719         static constexpr std::size_t bdryLevel = numLevels-2;
720         /// Index of all simplex dimensions in the complex.
721         using LevelIndex = typename std::make_index_sequence<numLevels>;
722     private:
723         /// Alias templated asc_node<...> as Node<k>
724         template <std::size_t k> using Node = detail::asc_node<KeyType, k, topLevel, NodeDataTypes,
EdgeDataTypes>;
725         /// Alias Node<k>* as NodePtr<k>
726         template <std::size_t k> using NodePtr = Node<k>*;
727
728     public:
729         /** Convenience alias for the user specified NodeData<k> typename */
730         template <std::size_t k> using NodeData = typename util::type_get<k, NodeDataTypes>::type;
731         /** Convenience alias for the user specified EdgeData<k> typename */
732         template <std::size_t k> using EdgeData = typename util::type_get<k, EdgeDataTypes>::type;
733
734         friend struct SimplexID; /**< SimplexID is a friend of
735                                 simplicial_complex */
736
737         /**
738          * @brief      A handle for a simplex object in the complex.
739          *
740          * SimplexID wraps a Node* for external handling. This way
741          * the end users are never exposed to a raw pointer. For all general
742          * purposes algorithms should use and pass SimplexIDs over raw pointers.
743          *
744          * @tparam      k      The Simplex dimension.
745          */
746         template <std::size_t k>
747         struct SimplexID {
748             /// Typename of the complex
749             using complex = simplicial_complex<traits>;
750             /// SimplexID is a friend of the complex
751             friend simplicial_complex<traits>;
752             /// The dimension of the simplex.
753             static constexpr std::size_t level = k;
754
755             /**
756              * @brief      Default constructor wraps a nullptr.
757              */
758             SimplexID() : ptr(nullptr) {}
759
760             /**
761              * @brief      Constructor to wrap a NodePtr<k>.
762              *
763              * @param[in]  p      The NodePtr to wrap
764              */
765             SimplexID(NodePtr<k> p) : ptr(p) {}
766
767             /**
768              * @brief      Copy constructor.
769              *
770              * @param[in]  rhs    Another SimplexID to copy.
771              */
772             SimplexID(const SimplexID &rhs) : ptr(rhs.ptr) {}
773
774             /// Assignment operator
775             SimplexID &operator=(const SimplexID &rhs) { ptr = rhs.ptr; return *this; }
776
777             /// Equality of wrapped pointers

```



```

778     friend bool operator==(SimplexID lhs, SimplexID rhs) { return lhs.ptr == rhs.ptr; }
779     /// Inequality of wrapped pointers
780     friend bool operator!=(SimplexID lhs, SimplexID rhs) { return lhs.ptr != rhs.ptr; }
781     /// Compare wrapped pointers
782     friend bool operator<=(SimplexID lhs, SimplexID rhs) { return lhs.ptr <= rhs.ptr; }
783     /// Compare wrapped pointers
784     friend bool operator>=(SimplexID lhs, SimplexID rhs) { return lhs.ptr >= rhs.ptr; }
785     /// Compare wrapped pointers
786     friend bool operator<(SimplexID lhs, SimplexID rhs) { return lhs.ptr < rhs.ptr; }
787     /// Compare wrapped pointers
788     friend bool operator>(SimplexID lhs, SimplexID rhs) { return lhs.ptr > rhs.ptr; }
789
790     /// Support casting to uintptr_t for hashing.
791     explicit operator std::uintptr_t () const { return reinterpret_cast<std::uintptr_t>(ptr); }
792
793     /// Dereferencing a SimplexID returns the data stored.
794     complex::NodeData<k> const &operator*() const { return ptr->_data; }
795     /// Dereferencing a SimplexID returns the data stored.
796     complex::NodeData<k> &operator*() { return ptr->_data; }
797
798     /// Get a handle to the stored data.
799     complex::NodeData<k> const &data() const { return ptr->_data; }
800     /// Get a handle to the stored data.
801     complex::NodeData<k> &data() { return ptr->_data; }
802
803     /**
804      * @brief      Gets the name of a simplex as an std::Array.
805      *
806      * @param[in]  id      SimplexID of the simplex of interest.
807      *
808      * @return      Array containing the name of 'id'.
809      */
810     std::array<KeyType, k> indices() const
811     {
812         std::array<KeyType, k> s;
813         std::size_t i = 0;
814         for (auto curr : ptr->_down)
815         {
816             s[i++] = curr.first;
817         }
818
819         return std::move(s);
820     }
821
822     // Valid in C++17
823     // TODO: (0) expose this to modern compilers
824     // if constexpr (k < complex::topLevel){
825     /**
826      * @brief      Insert the coboundary keys of a simple into an inserter.
827      *
828      * @param[in]  pos      Iterator inserter
829      *
830      * @tparam      Inserter  Typename of the inserter.
831      */
832     template <class Inserter>
833     void cover_insert(Inserter pos) const
834     {
835         for (auto curr : ptr->_up)
836         {
837             *pos++ = curr.first;
838         }
839     }
840
841     /**
842      * @brief      Get the coboundary keys of a simplex.
843      *
844      * @return      A vector of coboundary indices.
845      */
846     std::vector<KeyType> cover() const
847     {
848         std::vector<KeyType> rval;
849         cover_insert(std::back_inserter(rval));
850         return rval;
851     }
852 // }
853
854     /**
855      * @brief      Get a coboundary simplex
856      *
857      * @param[in]  s      Array of keys to follow
858      *
859      * @tparam      j      Number of keys
860      *
861      * @return      The simplex up
862      */
863     template <std::size_t j>
864     SimplexID<k+j> get_simplex_up(const KeyType (&s)[j]) const

```

```

865     {
866         static_assert(k+j <= complex::topLevel, "Cannot get simplex greater than the facets");
867         return complex::get_recurse<k, j>::apply(s, this->ptr);
868     }
869
870     /**
871     * @brief      Get a coboundary simplex
872     *
873     * @param[in]  arr    Array of keys to follow
874     *
875     * @tparam     j      Number of keys
876     *
877     * @return     The simplex up
878     */
879     template <std::size_t j>
880     SimplexID<k+j> get_simplex_up(const std::array<KeyType, j> &arr) const
881     {
882         static_assert(k+j <= complex::topLevel, "Cannot get simplex greater than the facets");
883         return get_recurse<k, j>::apply(arr.data(), this->ptr);
884     }
885
886     /**
887     * @brief      Convenience version of get_simplex_up when the name 's'
888     *              consists of a single character.
889     *
890     * @param[in]  id      The identifier of a simplex.
891     * @param[in]  s        The relative single character name of the desired
892     *                      simplex.
893     *
894     * @tparam     i        The size of simplex 'id'.
895     *
896     * @return     SimplexID of node corresponding to \f$id\cup s\f$.
897     */
898     SimplexID<k+1> get_simplex_up(const KeyType s) const
899     {
900         return get_recurse<k, 1>::apply(&s, this->ptr);
901     }
902
903     /**
904     * @brief      Gets the simplex down.
905     */
906     template <std::size_t j>
907     SimplexID<k-j> get_simplex_down(const KeyType (&s)[j]) const
908     {
909         return get_down_recurse<k, j>::apply(s, this->ptr);
910     }
911
912     /**
913     * @brief      Gets the simplex down.
914     */
915     template <std::size_t j>
916     SimplexID<k-j> get_simplex_down(const std::array<KeyType, j> &arr) const
917     {
918         return get_down_recurse<k, j>::apply(arr.data(), this->ptr);
919     }
920
921     /**
922     * @brief      Gets the simplex down.
923     */
924     SimplexID<k-1> get_simplex_down(const KeyType s) const
925     {
926         return get_down_recurse<k, 1>::apply(&s, this->ptr);
927     }
928
929     /**
930     * @brief      Print the simplex as its name.
931     *
932     * @param      out      Handle to the stream
933     * @param[in]  nid      SimplexID of interest
934     *
935     * @return     Handle to the stream
936     *
937     * Example
938     * ~~~~~~(.c)
939     * mesh.insert<3>({0,1,2});
940     * std::cout << s << std::endl;
941     * s{0,1,2}"
942     * ~~~~~~
943     */
944     friend std::ostream &operator<<(std::ostream &out,
945                                     const SimplexID &nid)
946     {
947         // currently no such thing as static_if in c++ so we use a
948         // template
949         // helper
950         out << "s{";
951         print_helper<k, 0>::apply(out, nid);

```

```

952         out << " ";
953         return out;
954     }
955
956
957     // NOTE: Manually swap out these print functions for debugging if
958     // desired.
959     // /**
960     //  * @brief      A full debug printout of of the node itself
961     //  *
962     //  * @param      out      Handle to the stream
963     //  * @param[in]   nid      SimplexID of interest
964     //  *
965     //  * @return     Handle to the stream
966     //  */
967     // friend std::ostream& operator<<(std::ostream& out, const
968     // SimplexID& nid){ out << *nid.ptr; return out; }
969
970     // /**
971     //  * @brief      Print the SimplexID as an ID.
972     //  *
973     //  * Example "0x7fd502402f10"
974     //  *
975     //  * @param      out      Handle to the stream
976     //  * @param[in]   nid      Node of interest
977     //  *
978     //  * @return     Handle to the stream
979     //  */
980     // friend std::ostream& operator<<(std::ostream& out, const
981     // SimplexID& nid) { out << nid.ptr; return out; }
982
983 private:
984     /**
985     * @brief      Base Case helper for printing SimplexIDs.
986     *
987     * @tparam      l          The Simplex dimension
988     * @tparam      foo        Dummy argument to avoid explicit
989     * specialization
990     *
991     * in class scope
992     */
993     template <std::size_t l, std::size_t foo>
994     struct print_helper
995     {
996         /**
997         * @brief      Print out the name of the simplex.
998         *
999         * @param      out      Stream to pipe to.
1000         * @param[in]   nid      The simplex to print.
1001         *
1002         * @return     Handle to the output stream.
1003         */
1004         static std::ostream& apply(std::ostream& out,
1005                                     const SimplexID<l> &nid)
1006         {
1007             auto down = (*nid.ptr)._down;
1008             for (auto it = down.cbegin(); it != down.cend()-1; ++it)
1009             {
1010                 out << it->first << ", ";
1011             }
1012             out << (down.cend()-1)->first;
1013             return out;
1014         }
1015     };
1016
1017     /**
1018     * @brief      Explicit specialization to print 0-Simplices
1019     *
1020     * @tparam      foo        Dummy argument to avoid explicit
1021     * specialization
1022     *
1023     * in class scope
1024     */
1025     template <std::size_t foo>
1026     struct print_helper<0, foo>
1027     {
1028         /**
1029         * @brief      Print the root simplex
1030         *
1031         * @param      out      Stream to print to.
1032         * @param[in]   nid      The simplex to print.
1033         *
1034         * @return     Handle to the output stream.
1035         */
1036         static std::ostream& apply(std::ostream& out,
1037                                     const SimplexID& nid)
1038         {
1039             out << "root " << nid;
1040             return out;
1041         }
1042     };

```

```

1039         }
1040     };
1041     /// The wrapped pointer.
1042     NodePtr<k> ptr;
1043 };
1044
1045 friend struct EdgeID; /**< EdgeID is a friend to simplicial_complex */
1046
1047 /**
1048  * @brief      External reference to an edge or a connection within the
1049  *              complex.
1050  *
1051  * @tparam      k      The edge connects a simplex of size k-1 to a
1052  *                      simplex of size k.
1053  */
1054 template <std::size_t k>
1055 struct EdgeID {
1056     /// Typename of the complex
1057     using complex = simplicial_complex<traits>;
1058     /// EdgeID is a friend of the complex
1059     friend simplicial_complex<traits>;
1060     /// The dimension of the simplex which the edge points to.
1061     static constexpr std::size_t level = k;
1062
1063     /**
1064      * @brief      Default constructor wraps a nullptr and dummy edge.
1065      */
1066     EdgeID() : ptr(nullptr), edge(0) {}
1067
1068     /**
1069      * @brief      Constructor to wrap an Edge.
1070      *
1071      * @param[in]   p      Pointer to the next Node.
1072      * @param[in]   e      Key of the edge
1073      */
1074     EdgeID(NodePtr<k> p, KeyType e) : ptr(p), edge(e) {}
1075
1076     /**
1077      * @brief      Copy constructor
1078      *
1079      * @param[in]   rhs     The right hand side
1080      */
1081     EdgeID(const EdgeID &rhs) : ptr(rhs.ptr), edge(rhs.edge) {}
1082
1083     /// Assignment operator
1084     EdgeID &operator=(const EdgeID &rhs) { ptr = rhs.ptr; edge = rhs.edge; return *this; }
1085
1086     /// Equality of wrapped pointers and edges
1087     friend bool operator==(EdgeID lhs, EdgeID rhs) { return lhs.ptr == rhs.ptr && lhs.edge ==
1088     rhs.edge; }
1089     /// Compare wrapped pointers and edges.
1089     friend bool operator!=(EdgeID lhs, EdgeID rhs) { return !(lhs == rhs); }
1090     /// Compare wrapped pointers and edges.
1091     friend bool operator<=(EdgeID lhs, EdgeID rhs) { return lhs < rhs || lhs == rhs; }
1092     /// Compare wrapped pointers and edges.
1093     friend bool operator>=(EdgeID lhs, EdgeID rhs) { return lhs > rhs || lhs == rhs; }
1094     /// Less than defines an ordering of key types on the edges.
1095     friend bool operator<(EdgeID lhs, EdgeID rhs)
1096     {
1097         return (lhs.ptr < rhs.ptr) || (lhs.ptr == rhs.ptr && lhs.edge < rhs.edge);
1098     }
1099     /// Greater than comparison
1100     friend bool operator>(EdgeID lhs, EdgeID rhs) { return rhs < lhs; }
1101
1102     // explicit operator std::size_t () const { return
1103     // static_cast<std::size_t>(ptr);
1104
1105     /// Dereferencing an EdgeID gets the data on the edge.
1106     auto const &operator*() const { return data(); }
1107     /// Dereferencing an EdgeID gets the data on the edge.
1108     auto &operator*() { return data(); }
1109
1110     /// Get the key of the edge.
1111     KeyType key() const { return edge; }
1112
1113     /// Return the data stored on the edge.
1114     auto const &data() const { return ptr->_edge_data[edge]; }
1115     /// Return the data stored on the edge.
1116     auto &data() { return ptr->_edge_data[edge]; }
1117
1118     /**
1119      * @brief      Get the coboundary simplex.
1120      *
1121      * @return      SimplexID of the simplex above the edge.
1122      */
1123     SimplexID<k> up() const { return ptr; }
1124

```

```

1125         /**
1126          * @brief      Get the simplex below.
1127          *
1128          * @return     SimplexID of the simplex below the edge.
1129          */
1130         SimplexID<k-1> down() const { return SimplexID<k-1>(ptr->_down[edge]); }
1131
1132     private:
1133         /// Pointer to the next node.
1134         NodePtr<k> ptr;
1135         /// The Key of the edge.
1136         KeyType edge;
1137     };
1138
1139     /**
1140     * @brief      Default constructor
1141     */
1142     simplicial_complex()
1143     : node_count(0)
1144     {
1145         for (auto &x : level_count) // Initialize level_count to 0 for all
1146             // levels
1147             {
1148                 x = 0;
1149             }
1150         // Create a root node
1151         _root = create_node<0>();
1152     }
1153
1154     /**
1155     * @brief      Destruct the simplicial complex.
1156     *
1157     * Recursively go over the simplices and remove them prior to
1158     * destructing
1159     * the CASC object itself.
1160     */
1161     ~simplicial_complex()
1162     {
1163         std::size_t count;
1164         remove_recurse<0, 0>::apply(this, &_root, &_root + 1, count);
1165     }
1166
1167     /**
1168     * @brief      Insert a simplex and all sub-simplices into the complex.
1169     *
1170     * Example -- insert the simplex {1,2,3}:
1171     * ~~~~~{.cpp}
1172     * mesh.insert<3>({1,2,3});
1173     * ~~~~~
1174     *
1175     * @param[in]  s      A C style array of vertices of simplex 's'.
1176     *
1177     * @tparam     n      Dimension of simplex 's'.
1178     */
1179     template <std::size_t n>
1180     SimplexID<n> insert(const KeyType (&s)[n])
1181     {
1182         for (const KeyType* p = s; p < s + n; ++p)
1183         {
1184             unused_vertices.remove(*p);
1185         }
1186         return insert_full<0, n>::apply(this, _root, s);
1187     }
1188
1189     /**
1190     * @brief      Insert a simplex and all sub-simplices into the complex
1191     *              along with data.
1192     *
1193     * Example -- insert the simplex {1,2,3} with data:
1194     * ~~~~~{.cpp}
1195     * mesh.insert<3>({1,2,3}, 5);
1196     * ~~~~~
1197     *
1198     * @param[in]  s      A C style array of vertices of simplex 's'.
1199     * @param[in]  data    The data to be stored at the simplex 's'.
1200     *
1201     * @tparam     n      Dimension of simplex 's'.
1202     */
1203     template <std::size_t n>
1204     SimplexID<n> insert(const KeyType (&s)[n], const NodeData<n> &data)
1205     {
1206         for (const KeyType* p = s; p < s + n; ++p)
1207         {
1208             unused_vertices.remove(*p);
1209         }
1210         Node<n>* rval = insert_full<0, n>::apply(this, _root, s);
1211         rval->_data = data;

```

```

1212         return rval;
1213     }
1214
1215     /**
1216     * @brief      Insert a simplex named and all sub-simplices into the
1217     * complex.
1218     *
1219     * @param[in]  s      Array of vertices comprising 's'.
1220     *
1221     * @tparam     n      Dimension of simplex 's'.
1222     */
1223     template <std::size_t n>
1224     SimplexID<n> insert(const std::array<KeyType, n> &s)
1225     {
1226         for (KeyType x : s)
1227         {
1228             unused_vertices.remove(x);
1229         }
1230         return insert_full<0, n>::apply(this, _root, s.data());
1231     }
1232
1233     /**
1234     * @brief      Insert a simplex and all sub-simplices into the complex
1235     * along with data.
1236     *
1237     * @param[in]  s      Array of vertices comprising 's'.
1238     * @param[in]  data   The data to be stored at the simplex 's'.
1239     *
1240     * @tparam     n      Dimension of simplex 's'.
1241     */
1242     template <std::size_t n>
1243     SimplexID<n> insert(const std::array<KeyType, n> &s, const NodeData<n> &data)
1244     {
1245         for (KeyType x : s)
1246         {
1247             unused_vertices.remove(x);
1248         }
1249         Node<n>* rval = insert_full<0, n>::apply(this, _root, s.data());
1250         rval->_data = data;
1251         return rval;
1252     }
1253
1254     /**
1255     * @brief      Add a new vertex to the complex.
1256     *
1257     * A list of currently unused indices are tracked using a B-tree. This
1258     * function retrieves a currently unused index and creates a new vertex
1259     * while returning the new key.
1260     *
1261     * @return     The key of the new vertex.
1262     */
1263     KeyType add_vertex()
1264     {
1265         KeyType v[1] = {unused_vertices.pop()};
1266         insert<1>(v);
1267         return v[0];
1268     }
1269
1270     /**
1271     * @brief      Add a new vertex to the complex with data.
1272     *
1273     * @return     The key of the new vertex.
1274     */
1275     KeyType add_vertex(const NodeData<1> &data)
1276     {
1277         KeyType v[1] = {unused_vertices.pop()};
1278         insert<1>(v, data);
1279         return v[0];
1280     }
1281
1282     /**
1283     * @brief      Apply a lambda function the name of a simplex.
1284     *
1285     * @param[in]  id      SimplexID of the simplex of interest.
1286     * @param[in]  fn      Lambda function to apply to the name of 'id'.
1287     *
1288     * @tparam     n      Dimension of simplex 'id'.
1289     * @tparam     Lambda  Functor which supports operator(KeyType).
1290     */
1291     template <std::size_t n, typename Lambda>
1292     void get_name(SimplexID<n> id, Lambda fn) const
1293     {
1294         for (auto curr : id.ptr->_down)
1295         {
1296             fn(curr.first);
1297         }
1298     }

```

```

1299     }
1300
1301     /**
1302     * @brief      Gets the name of a simplex as an std::Array.
1303     *
1304     * @param[in]  id      SimplexID of the simplex of interest.
1305     *
1306     * @tparam     n        Size of the simplex referenced by 'id'.
1307     *
1308     * @return     Array containing the name of 'id'.
1309     */
1310     template <std::size_t n>
1311     std::array<KeyType, n> get_name(SimplexID<n> id) const
1312     {
1313         std::array<KeyType, n> s;
1314         std::size_t i = 0;
1315         for (auto curr : id.ptr->_down)
1316         {
1317             s[i++] = curr.first;
1318         }
1319         assert(i == n);
1320         return s;
1321     }
1322
1323     /**
1324     * @brief      Gets the name of a simplex.
1325     *
1326     * This is the explicit specialization which handles the empty set
1327     * simplex.
1328     *
1329     * @param[in]  id      SimplexID of the simplex of interest.
1330     *
1331     * @return     Array containing the name of 'id'.
1332     */
1333     std::array<KeyType, 0> get_name(SimplexID<0>) const
1334     {
1335         std::array<KeyType, 0> name{};
1336         return name;
1337     }
1338
1339
1340     /**
1341     * @brief      Gets the simplex with name 's'.
1342     *
1343     * @param[in]  s        Name of the simplex to find.
1344     *
1345     * @tparam     n        Dimension of simplex s.
1346     *
1347     * @return     SimplexID of node corresponding to 's'.
1348     */
1349     template <std::size_t n>
1350     SimplexID<n> get_simplex_up(const KeyType (&s)[n]) const
1351     {
1352         return get_recurse<0, n>::apply(s, _root);
1353     }
1354
1355     template <std::size_t n>
1356     SimplexID<n> get_simplex_up(const std::array<KeyType, n> &arr) const
1357     {
1358         return get_recurse<0, n>::apply(arr.data(), _root);
1359     }
1360
1361     /**
1362     * @brief      Get the simplex identifier which has the name 's'
1363     *              relative to the simplex 'id'.
1364     *
1365     * @param[in]  id      The identifier of a simplex.
1366     * @param[in]  s        The relative name of the desired simplex.
1367     *
1368     * @tparam     i        The size of simplex 'id'.
1369     * @tparam     j        The length of the name 's'.
1370     *
1371     * @return     SimplexID of node corresponding to \f$id\cup s\f$.
1372     */
1373     template <std::size_t i, std::size_t j>
1374     SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const KeyType (&s)[j]) const
1375     {
1376         return get_recurse<i, j>::apply(s, id);
1377     }
1378
1379     template <std::size_t i, std::size_t j>
1380     SimplexID<i+j> get_simplex_up(const SimplexID<i> id, const std::array<KeyType, j> &arr) const
1381     {
1382         return get_recurse<i, j>::apply(arr.data(), id);
1383     }
1384
1385

```

```

1386     /**
1387      * @brief      Convenience version of get_simplex_up when the name 's'
1388      *             consists of a single character.
1389      *
1390      * @param[in]   id      The identifier of a simplex.
1391      * @param[in]   s        The relative single character name of the desired
1392      *                       simplex.
1393      *
1394      * @tparam      i        The size of simplex 'id'.
1395      *
1396      * @return      SimplexID of node corresponding to \f$id\cup s\f$.
1397      */
1398     template <std::size_t i>
1399     SimplexID<i+1> get_simplex_up(const SimplexID<i> id, const KeyType s) const
1400     {
1401         return get_recurse<i, 1>::apply(&s, id.ptr);
1402     }
1403
1404     /**
1405      * @brief      Get the root simplex.
1406      *
1407      * @return      The root simplex.
1408      */
1409     SimplexID<0> get_simplex_up() const
1410     {
1411         return _root;
1412     }
1413
1414
1415     /**
1416      * @brief      Get the sub-simplex of the simplex 'id' which does not
1417      *             have 's' in the name.
1418      *
1419      * @param[in]   id      The identifier of a simplex.
1420      * @param[in]   s        The relative name of the desired simplex.
1421      *
1422      * @tparam      i        The size of simplex 'id'.
1423      * @tparam      j        The length of the name 's'
1424      *
1425      * @return      The node down.
1426      */
1427     template <std::size_t i, std::size_t j>
1428     SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const KeyType (&s)[j]) const
1429     {
1430         return get_down_recurse<i, j>::apply(s, id.ptr);
1431     }
1432
1433     template <std::size_t i, std::size_t j>
1434     SimplexID<i-j> get_simplex_down(const SimplexID<i> id, const std::array<KeyType, j> &arr) const
1435     {
1436         return get_down_recurse<i, j>::apply(arr.data(), id.ptr);
1437     }
1438
1439     /**
1440      * @brief      Convenience version of get_simplex_down when the name 's'
1441      *             consists of a single character.
1442      *
1443      * @param[in]   id      The identifier of a simplex.
1444      * @param[in]   s        The relative single character name of the desired
1445      *                       simplex.
1446      *
1447      * @tparam      i        The size of simplex 'id'.
1448      *
1449      * @return      The node down.
1450      */
1451     template <std::size_t i>
1452     SimplexID<i-1> get_simplex_down(const SimplexID<i> id, const KeyType s) const
1453     {
1454         return get_down_recurse<i, 1>::apply(&s, id.ptr);
1455     }
1456
1457     /**
1458      * @brief      Get the root simplex.
1459      *
1460      * @return      The root simplex.
1461      */
1462     SimplexID<0> get_simplex_down() const
1463     {
1464         return _root;
1465     }
1466
1467     /**
1468      * @brief      Insert the coboundary keys of a simple into an inserter.
1469      *
1470      * @param[in]   id      The identifier of a simplex.
1471      * @param[in]   pos      Iterator inserter
1472      *

```



```

1473     * @tparam k          The dimension of the simplex.
1474     * @tparam Inserter   Typename of the inserter.
1475     */
1476     template <std::size_t k, class Inserter>
1477     void get_cover_insert(const SimplexID<k> id, Inserter pos) const
1478     {
1479         for (auto curr : id.ptr->_up)
1480         {
1481             *pos++ = curr.first;
1482         }
1483     }
1484
1485     /**
1486     * @brief      Apply a lambda function to the coboundary keys.
1487     *
1488     * @param[in]  id      The identifier
1489     * @param[in]  fn      The function
1490     *
1491     * @tparam k      The dimension of the simplex.
1492     * @tparam Lambda Typename of a functor which supports
1493     *               operator(KeyType).
1494     */
1495     template <std::size_t k, class Lambda>
1496     void get_cover(const SimplexID<k> id, Lambda fn) const
1497     {
1498         for (auto curr : id.ptr->_up)
1499         {
1500             fn(curr.first);
1501         }
1502     }
1503
1504     /**
1505     * @brief      Get the coboundary keys of a simplex.
1506     *
1507     * @param[in]  id      The identifier of a simplex.
1508     *
1509     * @tparam k      The dimension of the simplex.
1510     *
1511     * @return     A vector of coboundary indices.
1512     */
1513     template <std::size_t k>
1514     std::vector<KeyType> get_cover(const SimplexID<k> id) const
1515     {
1516         std::vector<KeyType> rval;
1517         get_cover_insert(id, std::back_inserter(rval));
1518         return rval;
1519     }
1520
1521     /**
1522     * @brief      Get the coboundary of a set of simplices.
1523     *
1524     * @param      simplices The set of simplices
1525     *
1526     * @tparam k      The dimension of the simplices.
1527     *
1528     * @return     The set of coboundary simplices.
1529     */
1530     template <std::size_t k>
1531     std::set<SimplexID<k+1> > up(const std::set<SimplexID<k> > &&simplices) const
1532     {
1533         std::set<SimplexID<k+1> > rval;
1534         for (auto simplex : simplices)
1535         {
1536             for (auto p : simplex.ptr->_up)
1537             {
1538                 rval.insert(SimplexID<k+1>(p.second));
1539             }
1540         }
1541         return rval;
1542     }
1543
1544     /**
1545     * @brief      Get the coboundary of a set of simplices.
1546     *
1547     * @param      simplices The set of simplices
1548     *
1549     * @tparam k      The dimension of the simplices.
1550     *
1551     * @return     The set of coboundary simplices.
1552     */
1553     template <std::size_t k>
1554     std::set<SimplexID<k+1> > up(const std::set<SimplexID<k> > &simplices) const
1555     {
1556         std::set<SimplexID<k+1> > rval;
1557         for (auto simplex : simplices)
1558         {
1559             for (auto p : simplex.ptr->_up)

```

```

1560         {
1561             rval.insert(SimplexID<k+1>(p.second));
1562         }
1563     }
1564     return rval;
1565 }
1566
1567 /**
1568  * @brief      Get the coboundary of a simplex.
1569  *
1570  * @param      nid    The simplex of interest
1571  *
1572  * @tparam     k      The dimension of the simplex.
1573  *
1574  * @return     Set of (k+1)-simplices of which 'nid' is a face of.
1575  */
1576 template <std::size_t k>
1577 std::set<SimplexID<k+1> > up(const SimplexID<k> nid) const
1578 {
1579     std::set<SimplexID<k+1> > rval;
1580     for (auto p : nid.ptr->_up)
1581     {
1582         rval.insert(SimplexID<k+1>(p.second));
1583     }
1584     return rval;
1585 }
1586
1587 template <std::size_t k, class InsertIter>
1588 void up(const std::set<SimplexID<k>&& simplices, InsertIter iter) const
1589 {
1590     for (auto simplex : simplices)
1591     {
1592         for (auto p : simplex.ptr->_up)
1593         {
1594             *iter++ = SimplexID<k+1>(p.second);
1595         }
1596     }
1597 }
1598
1599 template <std::size_t k, class InsertIter>
1600 void up(const std::set<SimplexID<k>&& simplices, InsertIter iter) const
1601 {
1602     for (auto simplex : simplices)
1603     {
1604         for (auto p : simplex.ptr->_up)
1605         {
1606             *iter++ = SimplexID<k+1>(p.second);
1607         }
1608     }
1609 }
1610
1611 template <std::size_t k, class InsertIter>
1612 void up(const SimplexID<k> simplex, InsertIter iter) const
1613 {
1614     for (auto p : simplex.ptr->_up)
1615     {
1616         *iter++ = SimplexID<k+1>(p.second);
1617     }
1618 }
1619
1620 /**
1621  * @brief      Get the boundary of a set of simplices.
1622  *
1623  * @param      simplices The set of simplicies.
1624  *
1625  * @tparam     k      The dimension of the simplices.
1626  *
1627  * @return     The set of boundary simplices.
1628  */
1629 template <std::size_t k>
1630 std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &&simplices) const
1631 {
1632     std::set<SimplexID<k-1> > rval;
1633     for (auto nid : simplices)
1634     {
1635         for (auto p : nid.ptr->_down)
1636         {
1637             rval.insert(SimplexID<k-1>(p.second));
1638         }
1639     }
1640     return rval;
1641 }
1642
1643 /**
1644  * @brief      Get the boundary of a set of simplices.
1645  *
1646  * @param      simplices The set of simplicies.

```

```

1647     *
1648     * @tparam    k        The dimension of the simplices.
1649     *
1650     * @return    The set of boundary simplices.
1651     */
1652     template <std::size_t k>
1653     std::set<SimplexID<k-1> > down(const std::set<SimplexID<k> > &simplices) const
1654     {
1655         std::set<SimplexID<k-1> > rval;
1656         for (auto simplex : simplices)
1657         {
1658             for (auto p : simplex.ptr->_down)
1659             {
1660                 rval.insert(SimplexID<k-1>(p.second));
1661             }
1662         }
1663         return rval;
1664     }
1665
1666     /**
1667     * @brief      Get the boundary of a simplex.
1668     *
1669     * @param      simplex  The simplex of interest.
1670     *
1671     * @tparam      k        The dimension of the simplex.
1672     *
1673     * @return      Set of (k-1)-simplices of which 'simplex' is a coface of.
1674     */
1675     template <std::size_t k>
1676     std::set<SimplexID<k-1> > down(const SimplexID<k> simplex) const
1677     {
1678         std::set<SimplexID<k-1> > rval;
1679         for (auto p : simplex.ptr->_down)
1680         {
1681             rval.insert(SimplexID<k-1>(p.second));
1682         }
1683         return rval;
1684     }
1685
1686     template <std::size_t k, class InsertIter>
1687     void down(const std::set<SimplexID<k>&& simplices, InsertIter iter) const{
1688         for (auto simplex : simplices)
1689         {
1690             for (auto p : simplex.ptr->_down)
1691             {
1692                 *iter++ = SimplexID<k-1>(p.second);
1693             }
1694         }
1695     }
1696
1697     template <std::size_t k, class InsertIter>
1698     void down(const std::set<SimplexID<k>&& simplices, InsertIter iter) const{
1699         for (auto simplex : simplices)
1700         {
1701             for (auto p : simplex.ptr->_down)
1702             {
1703                 *iter++ = SimplexID<k-1>(p.second);
1704             }
1705         }
1706     }
1707
1708     template <std::size_t k, class InsertIter>
1709     void down(const SimplexID<k> simplex, InsertIter iter) const{
1710         for (auto p : simplex.ptr->_down)
1711         {
1712             *iter++ = SimplexID<k-1>(p.second);
1713         }
1714     }
1715
1716     /**
1717     * @brief      Gets the edge up from a simplex.
1718     *
1719     * @param[in]  simplex  The simplex of interest.
1720     * @param[in]  a        Key of the edge to get.
1721     *
1722     * @tparam      k        The level of the simplex of interest
1723     *
1724     * @return      The edge up.
1725     */
1726     template <std::size_t k>
1727     EdgeID<k+1> get_edge_up(SimplexID<k> simplex, KeyType a)
1728     {
1729         return EdgeID<k+1>(simplex.ptr->_up.at(a), a);
1730     }
1731
1732     /**
1733     * @brief      Gets the edge down from a simplex.

```

```

1734     *
1735     * @param[in] simplex The simplex of interest.
1736     * @param[in] a      Key of the edge to get.
1737     *
1738     * @tparam k      The level of the simplex of interest
1739     *
1740     * @return      The edge down.
1741     */
1742     template <std::size_t k>
1743     EdgeID<k> get_edge_down(SimplexID<k> simplex, KeyType a)
1744     {
1745         return EdgeID<k>(simplex.ptr, a);
1746     }
1747
1748     /**
1749     * @brief      Gets the edge up from a simplex.
1750     *
1751     * @param[in] simplex The simplex of interest.
1752     * @param[in] a      Key of the edge to get.
1753     *
1754     * @tparam k      The level of the simplex of interest
1755     *
1756     * @return      The edge up.
1757     */
1758     template <std::size_t k>
1759     EdgeID<k+1> get_edge_up(SimplexID<k> simplex, KeyType a) const
1760     {
1761         return EdgeID<k+1>(simplex.ptr->_up.at(a), a);
1762     }
1763
1764     /**
1765     * @brief      Gets the edge down from a simplex.
1766     *
1767     * @param[in] simplex The simplex of interest.
1768     * @param[in] a      Key of the edge to get.
1769     *
1770     * @tparam k      The level of the simplex of interest
1771     *
1772     * @return      The edge down.
1773     */
1774     template <std::size_t k>
1775     EdgeID<k> get_edge_down(SimplexID<k> simplex, KeyType a) const
1776     {
1777         return EdgeID<k>(simplex.ptr, a);
1778     }
1779
1780     /**
1781     * @brief      Check whether a simplex with some name exists.
1782     *
1783     * @param[in] s      C-style array of the name
1784     *
1785     * @tparam k      The dimension of the simplex.
1786     *
1787     * @return      True if the simplex is in the complex.
1788     */
1789     template <std::size_t k>
1790     bool exists(const KeyType (&s)[k]) const
1791     {
1792
1793         return get_recurse<0, k>::apply(s, _root) != nullptr;
1794     }
1795
1796     /**
1797     * @brief      Get the number of simplices of dimension 'k'.
1798     *
1799     * @tparam k      The dimension of interest.
1800     *
1801     * @return      Integer number of k-simplices in the complex.
1802     */
1803     template <std::size_t k>
1804     std::size_t size() const
1805     {
1806         return std::get<k>(levels).size();
1807     }
1808
1809
1810     // template <std::size_t k> using SimplexIDIterator = detail::node_id_iterator<typename
1811     detail::map<NodePtr<k>::iterator, SimplexID<k>»;
1812     /**
1813     * @brief      Create an iterator to traverse the SimplexIDs of a
1814     *              dimension.
1815     *
1816     * @tparam k      The simplex dimension to traverse.
1817     *
1818     * @return      An iterator across all k-simplices of the complex.
1819     */
1820     template <std::size_t k>

```

```

1820     auto get_level_id()
1821     {
1822         auto begin = std::get<k>(levels).begin();
1823         auto end = std::get<k>(levels).end();
1824         auto data_begin = detail::make_node_id_iterator<decltype(begin), SimplexID<k> >(begin);
1825         auto data_end = detail::make_node_id_iterator<decltype(end), SimplexID<k> >(end);
1826         return util::make_range(data_begin, data_end);
1827     }
1828
1829     /**
1830      * @brief      Create an iterator to traverse the SimplexIDs of a
1831      *              dimension.
1832      *
1833      * @tparam      k      The simplex dimension to traverse.
1834      *
1835      * @return      An iterator across all k-simplices of the complex.
1836      */
1837     template <std::size_t k>
1838     auto get_level_id() const
1839     {
1840         auto begin = std::get<k>(levels).cbegin();
1841         auto end = std::get<k>(levels).cend();
1842         auto data_begin = detail::make_node_id_iterator<decltype(begin), const SimplexID<k>
1843 >(begin);
1844         auto data_end = detail::make_node_id_iterator<decltype(end), const SimplexID<k> >(end);
1845         return util::make_range(data_begin, data_end);
1846     }
1847
1848     // template <std::size_t k> using DataIterator = detail::node_data_iterator<typename
1849 std::map<std::size_t, NodePtr<k>>::iterator, NodeData<k>;
1850     /**
1851      * @brief      Create an iterator to traverse the simplex data of a
1852      *              dimension.
1853      *
1854      * @tparam      k      The simplex dimension to traverse.
1855      *
1856      * @return      An iterator across the data of all k-simplices in the
1857      *              complex.
1858      */
1859     template <std::size_t k>
1860     auto get_level()
1861     {
1862         auto begin = std::get<k>(levels).begin();
1863         auto end = std::get<k>(levels).end();
1864         auto data_begin = detail::make_node_data_iterator<decltype(begin), NodeData<k> >(begin);
1865         auto data_end = detail::make_node_data_iterator<decltype(end), NodeData<k> >(end);
1866         return util::make_range(data_begin, data_end);
1867     }
1868
1869     /**
1870      * @brief      Create an iterator to traverse the simplex data of a
1871      *              dimension.
1872      *
1873      * @tparam      k      The simplex dimension to traverse.
1874      *
1875      * @return      An iterator across the data of all k-simplices in the
1876      *              complex.
1877      */
1878     template <std::size_t k>
1879     auto get_level() const
1880     {
1881         auto begin = std::get<k>(levels).cbegin();
1882         auto end = std::get<k>(levels).cend();
1883         auto data_begin = detail::make_node_data_iterator<decltype(begin), const NodeData<k>
1884 >(begin);
1885         auto data_end = detail::make_node_data_iterator<decltype(end), const NodeData<k> >(end);
1886         return util::make_range(data_begin, data_end);
1887     }
1888
1889     /**
1890      * @brief      Remove a simplex and all dependent simplices by name.
1891      *
1892      * @param[in]   s      C-style array with the name of the simplex to
1893      *                      remove.
1894      *
1895      * @tparam      k      The dimension of the simplex.
1896      *
1897      * @return      Integer corresponding to the number of simplices removed.
1898      */
1899     template <std::size_t k>
1900     std::size_t remove(const KeyType (&s)[k])
1901     {
1902         Node<k>* root = get_recurse<0, k>::apply(s, _root);
1903         std::size_t count = 0;
1904         return remove_recurse<k, 0>::apply(this, &root, &root + 1, count);
1905     }

```

```

1904
1905 /**
1906  * @brief      Remove a simplex and all dependent simplices by name.
1907  *
1908  * @param[in]   s      std::array with the name of the simplex to remove.
1909  *
1910  * @tparam      k      The dimension of the simplex.
1911  *
1912  * @return      Integer corresponding to the number of simplices removed.
1913  */
1914 template <std::size_t k>
1915 std::size_t remove(const std::array<KeyType, k> &s)
1916 {
1917     Node<k>* root = get_recurse<k>, k>::apply(s.data(), _root);
1918     std::size_t count = 0;
1919     return remove_recurse<k>, 0>::apply(this, &root, &root + 1, count);
1920 }
1921
1922 /**
1923  * @brief      Remove a simplex and all dependent simplices by
1924  *              SimplexID.
1925  *
1926  * @param[in]   s      SimplexID of the simplex to remove.
1927  *
1928  * @tparam      k      The dimension of the simplex.
1929  *
1930  * @return      Integer corresponding to the number of simplices removed.
1931  */
1932 template <std::size_t k>
1933 std::size_t remove(SimplexID<k> s)
1934 {
1935     std::size_t count = 0;
1936     return remove_recurse<k>, 0>::apply(this, &s.ptr, &s.ptr + 1, count);
1937 }
1938
1939 /**
1940  * @brief      Checks whether a simplex is on a boundary.
1941  *
1942  * @param[in]   s      SimplexID of interest
1943  *
1944  * @tparam      k      Dimension of the simplex
1945  *
1946  * @return      True if the simplex is a member of a topLevel-1 simplex
1947  *              on the boundary or if the simplex is on a boundary or if
1948  *              the simplex is a coboundary of a boundary topLevel-1
1949  *              simplex.
1950  */
1951 template <std::size_t k>
1952 bool onBoundary(const SimplexID<k> s) const
1953 {
1954     return onBoundaryH<k>, 0>::apply(s);
1955 }
1956
1957
1958 /**
1959  * @brief      Checks whether a simplex is near a boundary.
1960  *
1961  * @param[in]   s      SimplexID of interest
1962  *
1963  * @tparam      level  Dimension of the simplex
1964  *
1965  * @return      True if the simplex or any subsimplices are onBoundary.
1966  */
1967 template <std::size_t level>
1968 bool nearBoundary(const SimplexID<level> s) const
1969 {
1970     return nearBoundaryH<level>, 0>::apply(s);
1971 }
1972
1973 /** Reintroduce this code block when this is resolved
1974  * http://www.open-std.org/jtc1/sc22/wg21/docs/cwg\_defects.html#727
1975  */
1976 // /**
1977 //  * @brief      Checks whether a simplex is on a boundary.
1978 //  *
1979 //  * @param[in]   s      SimplexID of interest
1980 //  *
1981 //  * @tparam      k      Dimension of the simplex
1982 //  *
1983 //  * @return      True if the simplex interacts with a
1984 //  *              topLevel-1 simplex which is on a boundary.
1985 //  */
1986 // template <std::size_t k>
1987 // bool onBoundary(const SimplexID<k> s) const
1988 // {
1989 //     for(auto p : s.ptr->_up)
1990 //         {

```

```

1991         //         if (onBoundary(SimplexID<k+1>(p.second)))
1992             //             return true;
1993         //     }
1994         //     return false;
1995     // }
1996
1997     // /**
1998     //  * @brief      Specialization of the facets
1999     //  *
2000     //  * @param[in]  s      SimplexID of interest
2001     //  *
2002     //  * @tparam     k      Dimension of the simplex
2003     //  *
2004     //  * @return     True if s is on a boundary
2005     //  */
2006     // template<>
2007     // bool onBoundary(const SimplexID<topLevel> s) const
2008     // {
2009     //     for(auto p : s.ptr->_down){
2010     //         if (onBoundary(SimplexID<topLevel-1>(p.second)))
2011     //             return true;
2012     //     }
2013     //     return false;
2014     // }
2015
2016     // /**
2017     //  * @brief      Specialization of the topLevel-1 simplices
2018     //  *
2019     //  * @param[in]  s      SimplexID of interest
2020     //  *
2021     //  * @tparam     k      Dimension of the simplex
2022     //  *
2023     //  * @return     True if s is on a boundary
2024     //  */
2025     // template<>
2026     // bool onBoundary(const SimplexID<topLevel-1> s) const
2027     // {
2028     //     return s.ptr->_up.size() != 2;
2029     // }
2030
2031
2032     /**
2033     * @brief      Less than or equal to comparison operator of two
2034     *              SimplexIDs.
2035     *
2036     * @param[in]  lhs    The left hand side
2037     * @param[in]  rhs    The right hand side
2038     *
2039     * @tparam     L      Dimension of lhs simplex.
2040     * @tparam     R      Dimension of rhs simplex.
2041     *
2042     * @return     True if lhs is rhs or a proper face of rhs.
2043     */
2044     template <std::size_t L, std::size_t R>
2045     bool leq(SimplexID<L> lhs, SimplexID<R> rhs) const
2046     {
2047         auto name_lhs = get_name(lhs);
2048         auto name_rhs = get_name(rhs);
2049
2050         std::size_t i = 0;
2051         for (std::size_t j = 0; i < L && j < R; ++j)
2052         {
2053             if (name_lhs[i] == name_rhs[j])
2054             {
2055                 ++i;
2056             }
2057         }
2058         return (i == L);
2059     }
2060
2061     /**
2062     * @brief      Equality comparison of two simplices.
2063     *
2064     * @param[in]  lhs    The left hand side
2065     * @param[in]  rhs    The right hand side
2066     *
2067     * @tparam     L      Dimension of lhs simplex.
2068     * @tparam     R      Dimension of rhs simplex.
2069     *
2070     * @return     Always false as L != R. The L==R case is overloaded by
2071     *              partial specialization.
2072     */
2073     template <std::size_t L, std::size_t R>
2074     bool eq(SimplexID<L>, SimplexID<R>) const
2075     {
2076         return false;
2077     }

```

```

2078
2079 /**
2080  * @brief      Equality comparison of two simplices.
2081  *
2082  * @param[in]   lhs    The left hand side
2083  * @param[in]   rhs    The right hand side
2084  *
2085  * @tparam      k      Dimension of the simplices.
2086  *
2087  * @return      True if the names are the same.
2088  */
2089 template <std::size_t k>
2090 bool eq(SimplexID<k> lhs, SimplexID<k> rhs) const
2091 {
2092     auto name_lhs = get_name(lhs);
2093     auto name_rhs = get_name(rhs);
2094
2095     for (std::size_t i = 0; i < k; ++i)
2096     {
2097         if (name_lhs[i] != name_rhs[i])
2098         {
2099             return false;
2100         }
2101     }
2102     return true;
2103 }
2104
2105 /**
2106  * @brief      Less than comparison of simplices.
2107  *
2108  * @param[in]   lhs    The left hand side
2109  * @param[in]   rhs    The right hand side
2110  *
2111  * @tparam      L      Dimension of lhs simplex.
2112  * @tparam      R      Dimension of rhs simplex.
2113  *
2114  * @return      True if lhs is a proper subface of rhs.
2115  */
2116 template <std::size_t L, std::size_t R>
2117 bool lt(SimplexID<L> lhs, SimplexID<R> rhs) const
2118 {
2119     return L < R && leq(lhs, rhs);
2120 }
2121
2122 private:
2123 /**
2124  * @brief      Base case for checking if simplex is near a boundary
2125  *
2126  * @tparam      level   Dimension of the simplex
2127  * @tparam      foo     Dummy argument to avoid explicit specialization in
2128  *                      class scope
2129  */
2130 template <std::size_t level, std::size_t foo>
2131 struct nearBoundaryH
2132 {
2133     static bool apply(const SimplexID<level> s){
2134         auto name = s.indices();
2135         KeyType down[level-1];
2136
2137         for(std::size_t i = 0; i < level; ++i){
2138             std::size_t k = 0;
2139             for(std::size_t j = 0; j < level; ++j){
2140                 if (i != j){
2141                     down[k++] = name[j];
2142                 }
2143             }
2144             if(onBoundaryH<1, 0>::apply(
2145                 get_down_recurse<level, level-1>::apply(down, s.ptr)
2146             ))
2147                 return true;
2148         }
2149         return false;
2150     }
2151 };
2152
2153 /**
2154  * @brief      Specialization of vertices
2155  *
2156  * @tparam      foo     Dummy argument to avoid explicit specialization in
2157  *                      class scope
2158  */
2159 template <std::size_t foo>
2160 struct nearBoundaryH<1, foo>
2161 {
2162     static bool apply(const SimplexID<1> s){
2163         if(onBoundaryH<1, 0>::apply(s))
2164             return true;

```



```

2165         return false;
2166     }
2167 };
2168
2169 /**
2170  * @brief      Base case for checking if simplex is on a boundary
2171  *
2172  * @tparam      level      Dimension of the simplex
2173  * @tparam      foo        Dummy argument to avoid explicit specialization in
2174  *                          class scope
2175  */
2176 template <std::size_t level, std::size_t foo>
2177 struct onBoundaryH
2178 {
2179     /**
2180      * @brief      Recurse up complex to find boundary.
2181      *
2182      * @param[in]   s        Simplex of interest
2183      *
2184      * @return      True if on boundary
2185      */
2186     static bool apply(const SimplexID<level> s)
2187     {
2188         for(auto p : s.ptr->_up)
2189         {
2190             if(onBoundaryH<level+1, foo>::apply(SimplexID<level+1>(p.second)))
2191                 return true;
2192         }
2193         return false;
2194     }
2195 };
2196
2197 /**
2198  * @brief      Specialization for if facets are on boundary.
2199  *
2200  * @tparam      foo        Dummy argument to avoid explicit specialization in
2201  *                          class scope
2202  */
2203 template <std::size_t foo>
2204 struct onBoundaryH<topLevel, foo>
2205 {
2206     /**
2207      * @brief      Check if a face is on a boundary
2208      *
2209      * @param[in]   s        SimplexID<topLevel> of interest
2210      *
2211      * @return      True if a member SimplexID<topLevel-1> is a boundary.
2212      */
2213     static bool apply(const SimplexID<topLevel> s)
2214     {
2215         for(auto p : s.ptr->_down){
2216             if(onBoundaryH<topLevel-1, foo>::apply(SimplexID<topLevel-1>(p.second)))
2217                 return true;
2218         }
2219         return false;
2220     }
2221 };
2222
2223 /**
2224  * @brief      Specialization for topLevel-1 simplices
2225  *
2226  * @tparam      foo        Dummy argument to avoid explicit specialization in
2227  *                          class scope
2228  */
2229 template <std::size_t foo>
2230 struct onBoundaryH<bdryLevel, foo>
2231 {
2232     /**
2233      * @brief      Check if SimplexID<topLevel-1> is on a boundary
2234      *
2235      * @param[in]   s        SimplexID of interest
2236      *
2237      * @return      True if simplex has less than 2 coboundary faces.
2238      */
2239     static bool apply(const SimplexID<bdryLevel> s)
2240     {
2241         return s.ptr->_up.size() < 2;
2242     }
2243 };
2244
2245 /**
2246  * @brief      Base case for recursively deleting simplices.
2247  *
2248  * @tparam      level      Simplex dimension to operate at.
2249  * @tparam      foo        Dummy argument to avoid explicit specialization in
2250  *                          class scope
2251  */

```

```

2252     template <std::size_t level, std::size_t foo>
2253     struct remove_recurse
2254     {
2255         /**
2256          * @brief      Recursively remove simplices.
2257          *
2258          * @param      that    The CASC object
2259          * @param[in]   begin    Iterator to beginning of the set of simplices
2260          *              to remove.
2261          * @param[in]   end      Iterator to the end of the set.
2262          * @param      count    Number of simplices removed already.
2263          *
2264          * @tparam      T        Typename of the iterator.
2265          *
2266          * @return      Recurse to the next level and remove coboundary
2267          *              simplices.
2268          */
2269         template <typename T>
2270         static std::size_t apply(type_this* that, T begin, T end, std::size_t &count)
2271         {
2272             std::set<Node<level+1>*> next;
2273             // for each node of interest...
2274             for (auto i = begin; i != end; ++i)
2275             {
2276                 auto up = (*i)->_up;
2277                 for (auto j = up.begin(); j != up.end(); ++j)
2278                 {
2279                     next.insert(j->second);
2280                 }
2281                 that->remove_node(*i);
2282                 ++count;
2283             }
2284             return remove_recurse<level+1, foo>::apply(that, next.begin(), next.end(), count);
2285         }
2286     };
2287
2288     /**
2289     * @brief      Terminal condition for remove_recurse.
2290     *
2291     * @tparam      foo        Dummy argument to avoid explicit specialization in
2292     *                          class scope
2293     */
2294     template <std::size_t foo>
2295     struct remove_recurse<topLevel, foo>
2296     {
2297         /**
2298          * @brief      Remove the facets of the complex.
2299          *
2300          * @param      that    The CASC object
2301          * @param[in]   begin    Iterator to beginning of the set of simplices
2302          *              to remove.
2303          * @param[in]   end      Iterator to the end of the set.
2304          * @param      count    Number of simplices removed already.
2305          *
2306          * @tparam      T        Typename of the iterator.
2307          *
2308          * @return      The number of simplices removed
2309          */
2310         template <typename T>
2311         static std::size_t apply(type_this* that, T begin, T end, std::size_t &count)
2312         {
2313             for (auto i = begin; i != end; ++i)
2314             {
2315                 that->remove_node(*i);
2316                 ++count;
2317             }
2318             return count;
2319         }
2320     };
2321
2322     /**
2323     * @brief      Recursively retrieve a simplex of interest.
2324     *
2325     * @tparam      level    The current simplex dimension.
2326     * @tparam      n        Number of remaining times to recurse.
2327     */
2328     template <std::size_t level, std::size_t n>
2329     struct get_recurse
2330     {
2331         /**
2332          * @brief      Get the simplex of interest.
2333          *
2334          * @param[in]   that    The simplicial complex to search.
2335          * @param[in]   s        Pointer to an array of Keys.
2336          * @param      root    The current simplex
2337          *
2338          * @return      Returns a pointer to the node.

```

```

2339         */
2340     static Node<level+n>* apply(const KeyType* s, Node<level>* root)
2341     {
2342         // TODO: We probably don't need to check if root is a valid
2343         // simplex (10)
2344         if (root)
2345         {
2346             auto p = root->_up.find(*s);
2347             if (p != root->_up.end())
2348             {
2349                 return get_recurse<level+1, n-1>::apply(s+1, root->_up.at(*s));
2350             }
2351             else
2352             {
2353                 return nullptr;
2354             }
2355         }
2356         else
2357         {
2358             return nullptr;
2359         }
2360     }
2361 };
2362 /**
2363  * @brief      Recursively retrieve a simplex of interest.
2364  *
2365  * @tparam      level  The current simplex dimension.
2366  */
2367 template <std::size_t level>
2368 struct get_recurse<level, 0>
2369 {
2370     /**
2371     * @brief      Get the simplex of interest.
2372     *
2373     * @param[in]  that  The simplicial complex to search.
2374     * @param[in]  s      Pointer to an array of Keys.
2375     * @param      root   The current simplex
2376     *
2377     * @return      Returns a pointer to the node.
2378     */
2379     static Node<level>* apply(const KeyType*, Node<level>* root)
2380     {
2381         return root;
2382     }
2383 };
2384
2385 /**
2386  * @brief      Recursively retrieve a simplex of interest going down.
2387  *
2388  * @tparam      level  The current simplex dimension.
2389  * @tparam      n      Number of remaining times to recurse.
2390  */
2391 template <std::size_t level, std::size_t n>
2392 struct get_down_recurse
2393 {
2394     /**
2395     * @brief      Get the simplex of interest.
2396     *
2397     * @param[in]  that  The simplicial complex to search.
2398     * @param[in]  s      Pointer to an array of Keys.
2399     * @param      root   The current simplex
2400     *
2401     * @return      Returns a pointer to the node.
2402     */
2403     static Node<level-n>* apply(const KeyType* s, Node<level>* root)
2404     {
2405         if (root)
2406         {
2407             auto p = root->_down.find(*s);
2408             if (p != root->_down.end())
2409             {
2410                 return get_down_recurse<level-1, n-1>::apply(s+1, root->_down[*s]);
2411             }
2412             else
2413             {
2414                 return nullptr;
2415             }
2416         }
2417         else
2418         {
2419             return nullptr;
2420         }
2421     }
2422 };
2423
2424 /**
2425  * @brief      Recursively retrieve a simplex of interest going down.

```

```

2426     *
2427     * @tparam    level    The current simplex dimension.
2428     */
2429     template <std::size_t level>
2430     struct get_down_recurse<level, 0>
2431     {
2432         /**
2433          * @brief      Get the simplex of interest.
2434          *
2435          * @param[in]  this    The simplicial complex to search.
2436          * @param[in]  s      Pointer to an array of Keys.
2437          * @param      root    The current simplex
2438          *
2439          * @return      Returns a pointer to the node.
2440          */
2441         static Node<level>* apply(const KeyType*, Node<level>* root)
2442         {
2443             return root;
2444         }
2445     };
2446
2447     /**
2448     * @brief      Insert a simplex and all dependent simplices into the
2449     *             complex.
2450     *
2451     * @tparam      level    Dimension of the current root simplex
2452     * @tparam      n        The number of times to recurse.
2453     */
2454     template <std::size_t level, std::size_t n>
2455     struct insert_full
2456     {
2457         /**
2458          * @brief      Kick off a for loop to insert all cofaces.
2459          *
2460          * @param      that    The simplicial complex
2461          * @param      root    The current simplex to insert at.
2462          * @param[in]  begin    Pointer to an array of Keys.
2463          *
2464          * @return      Returns the node to insert.
2465          */
2466         static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
2467         {
2468             return insert_for<level, n, n>::apply(that, root, begin);
2469         }
2470     };
2471
2472
2473     /**
2474     * @brief      Insert a simplex and all dependent simplices into the
2475     *             complex.
2476     *
2477     * @tparam      level    Dimension of the current root simplex
2478     */
2479     template <std::size_t level>
2480     struct insert_full<level, 0>
2481     {
2482         /**
2483          * @brief      Terminal case.
2484          *
2485          * @param      that    The simplicial complex
2486          * @param      root    The current simplex to insert at.
2487          * @param[in]  begin    Pointer to an array of Keys.
2488          *
2489          * @return      Returns the node to insert.
2490          */
2491         static Node<level>* apply(type_this*, Node<level>* root, const KeyType*)
2492         {
2493             return root;
2494         }
2495     };
2496
2497     /**
2498     * @brief      Iterate over antistep
2499     *
2500     * @tparam      level      Dimension of the current root simplex
2501     * @tparam      antistep    Antistep to track which indices to append to root.
2502     * @tparam      n          Original antistep.
2503     */
2504     template <std::size_t level, std::size_t antistep, std::size_t n>
2505     struct insert_for
2506     {
2507         /**
2508          * @brief      Call insert_raw and continue for loop
2509          *
2510          * @param      that    The simplicial complex
2511          * @param      root    The current simplex to insert at.
2512          * @param[in]  begin    Pointer to an array of Keys.

```

```

2513         *
2514         * @return      Returns the node to insert.
2515         */
2516         static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
2517         {
2518             insert_raw<level, n-antistep>::apply(that, root, begin);
2519             return insert_for<level, antistep-1, n>::apply(that, root, begin);
2520         }
2521     };
2522
2523     /**
2524     * @brief          Terminal case.
2525     *
2526     * @tparam         level  Dimension of the current root simplex.
2527     * @tparam         n      Original antistep.
2528     */
2529     template <std::size_t level, std::size_t n>
2530     struct insert_for<level, 1, n>
2531     {
2532         /**
2533         * @brief          Call insert_raw and stop loop
2534         *
2535         * @param         that  The simplicial complex
2536         * @param         root  The current simplex to insert at.
2537         * @param[in]     begin  Pointer to an array of Keys.
2538         *
2539         * @return        Returns the node to insert.
2540         */
2541         static Node<level+n>* apply(type_this* that, Node<level>* root, const KeyType* begin)
2542         {
2543             return insert_raw<level, n-1>::apply(that, root, begin);
2544         }
2545     };
2546
2547     /**
2548     * @brief          Actually insert the node and connect up and down.
2549     *
2550     * @tparam         level  Dimension of the current root simplex.
2551     * @tparam         n      The index to append to root.
2552     */
2553     template <std::size_t level, std::size_t n>
2554     struct insert_raw
2555     {
2556         /**
2557         * @brief          Create the node and connect up and down.
2558         *
2559         * @param         that  The simplicial complex
2560         * @param         root  The current simplex to insert at.
2561         * @param[in]     begin  Pointer to an array of Keys.
2562         *
2563         * @return        Returns the node to insert.
2564         */
2565         static Node<level+n+1>* apply(type_this* that, Node<level>* root, const KeyType* begin)
2566         {
2567             KeyType v = *(begin+n);
2568             Node<level+1>* nn;
2569             // if root->v doesn't exist then create it
2570             auto iter = root->_up.find(v);
2571             if (iter == root->_up.end())
2572             {
2573                 nn = that->create_node<level+1>();
2574
2575                 nn->_down[v] = root;
2576                 root->_up[v] = nn;
2577                 that->backfill(root, nn, v);
2578             }
2579             else
2580             {
2581                 nn = iter->second; // otherwise get it
2582             }
2583             return insert_full<level+1, n>::apply(that, nn, begin);
2584         }
2585     };
2586
2587     /**
2588     * @brief          Backfill in the pointers from prior nodes to the new node
2589     *
2590     * @param         root  is a parent node
2591     * @param         nn    is the new child node
2592     * @param         value  is the exposed id of nn
2593     * @return        void
2594     *
2595     * @tparam         level  Dimension of the current root simplex.
2596     */
2597     template <std::size_t level>
2598     void backfill(Node<level>* root, Node<level+1>* nn, KeyType value)
2599 
```

```

2600     {
2601         for (auto curr = root->_down.begin(); curr != root->_down.end(); ++curr)
2602         {
2603             int v = curr->first;
2604
2605             Node<level-1>* parent = curr->second;
2606             Node<level> * child = parent->_up[value];
2607
2608             nn->_down[v] = child;
2609             child->_up[v] = nn;
2610         }
2611     }
2612
2613     /**
2614     * @brief Fill in the pointers from level 1 to 0.
2615     *
2616     * @param root is a level 0 node
2617     * @param nn is a level 1 node
2618     * @param value is the exposed id of nn
2619     * @return void
2620     */
2621     void backfill(Node<0>*, Node<1>*, int)
2622     {
2623         return;
2624     }
2625
2626     /**
2627     * @brief Creates a new node of some dimension.
2628     *
2629     * @param[in] x Argument to help deduce the new node dimension
2630     *
2631     * @tparam level Simplex dimension
2632     *
2633     * @return A pointer to the new node.
2634     */
2635     template <std::size_t level>
2636     Node<level>* create_node()
2637     {
2638         // Create the new node
2639         auto p = new Node<level>(node_count++);
2640         ++(level_count[level]); // Increment the count in the level
2641
2642         // node_count-1 to match the internal IDs correctly.
2643         MAYBE_UNUSED bool ret = std::get<level>(levels).insert(
2644             std::pair<std::size_t, NodePtr<level> >(node_count-1, p)).second;
2645         assert(ret);
2646         /*
2647         // sanity check to make sure there aren't duplicate keys...
2648         if (ret==false) {
2649             std::cout << "Error: Node '" << node_count << "' already existed
2650                 with value " << *p << std::endl;
2651         }
2652         */
2653         return p;
2654     }
2655
2656     /**
2657     * @brief Removes a node.
2658     *
2659     * @param p Simplex to remove
2660     *
2661     * @tparam level Dimension of the simplex
2662     */
2663     template <std::size_t level>
2664     void remove_node(Node<level>* p)
2665     {
2666         for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
2667         {
2668             curr->second->_up.erase(curr->first);
2669         }
2670         for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
2671         {
2672             curr->second->_down.erase(curr->first);
2673         }
2674         --(level_count[level]);
2675         std::get<level>(levels).erase(p->_node);
2676         delete p;
2677     }
2678
2679     /**
2680     * @brief Removes a node.
2681     *
2682     * @param p Simplex to remove
2683     */
2684     void remove_node(Node<1>* p)
2685     {
2686         // This for loop should only have a single iteration.

```

```

2687         for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
2688         {
2689             unused_vertices.insert(curr->first);
2690             curr->second->_up.erase(curr->first);
2691         }
2692         for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
2693         {
2694             curr->second->_down.erase(curr->first);
2695         }
2696         --(level_count[1]);
2697         std::get<1>(levels).erase(p->_node);
2698         delete p;
2699     }
2700
2701     /**
2702     * @brief      Removes a node.
2703     *
2704     * @param      p      Simplex to remove
2705     */
2706     void remove_node(Node<0>* p)
2707     {
2708         for (auto curr = p->_up.begin(); curr != p->_up.end(); ++curr)
2709         {
2710             curr->second->_down.erase(curr->first);
2711         }
2712         --(level_count[0]);
2713         std::get<0>(levels).erase(p->_node);
2714         delete p;
2715     }
2716
2717     /**
2718     * @brief      Removes a node.
2719     *
2720     * @param      p      Simplex to remove
2721     */
2722     void remove_node(Node<topLevel>* p)
2723     {
2724         for (auto curr = p->_down.begin(); curr != p->_down.end(); ++curr)
2725         {
2726             curr->second->_up.erase(curr->first);
2727         }
2728         --(level_count[topLevel]);
2729         std::get<topLevel>(levels).erase(p->_node);
2730         delete p;
2731     }
2732
2733     /// The root node
2734     NodePtr<0> _root;
2735     /// A counter of the total number of nodes.
2736     std::size_t node_count;
2737     /// A counter of the number of simplices per level.
2738     std::array<std::size_t, numLevels> level_count;
2739     /// Typename of a tuple of LevelIndex broadcasted with NodePtr<k>.
2740     using NodePtrLevel = typename util::int_type_map<std::size_t, std::tuple, LevelIndex,
2741     NodePtr>::type;
2742     /// Typename of a map of levels to NodePtr<k>'s.
2743     typename util::type_map<NodePtrLevel, detail::map>::type levels;
2744     /// B-tree of unused vertex indices.
2745     index_tracker::index_tracker<KeyType> unused_vertices;
2746 };
2747
2748
2749 /**
2750 * Alias to generate a CASC from a list of traits.
2751 * See also simplicial_complex_traits_default. Example -- To create a
2752 * tetrahedral mesh with integer data on all simplices:
2753 * ~~~~~{.cpp}
2754 * auto mesh = AbstractSimplicialComplex<
2755 *     int, // KEYTYPE
2756 *     int, // Root data
2757 *     int, // Vertex data
2758 *     int, // Edge data
2759 *     int, // Face data
2760 *     int  // Volume data
2761 * >();
2762 * ~~~~~
2763 */
2764 template <typename KeyType, typename ... Ts>
2765 using AbstractSimplicialComplex = simplicial_complex<
2766     detail::simplicial_complex_traits_default<KeyType, Ts...> >;
2767
2768 /// @cond detail
2769 namespace simplex_set_detail{
2770 /**
2771 * @brief      Template to compute a hash for a SimplexID.
2772 *

```

```

2773 * Since SimplexID is actually a wrapper around a Node* we have to hash it
2774 * accordingly. The static_cast calls the defined explicit operator which
2775 * reinterprets the stored Node* pointer as a uintptr_t which we can hash
2776 * directly.
2777 *
2778 * @tparam      SimplexID  Typename of the SimplexID.
2779 */
2780 template <typename SimplexID>
2781 struct hashSimplexID{
2782     /**
2783      * @brief    Compute the hash.
2784      *
2785      * ~~~~~ (.cpp)
2786      * std::cout << hashSimplexID<decltype(nid)>{}(nid) << std::endl;
2787      * ~~~~~
2788      *
2789      * @param[in]  nid      The simplex of interest.
2790      * @return     Resultant hash.
2791      */
2792     std::size_t operator()(const SimplexID nid) const
2793     {
2794         return std::hash<std::uintptr_t>() (static_cast<uintptr_t>(nid));
2795     }
2796 };
2797 } // end namespace simplex_set_detail
2798 /// @endcond
2799
2800 /// Helpful alias defining a unordered_set of simplices. See also hashSimplexID.
2801 template <typename T> using NodeSet =
2802     std::unordered_set<T, simplex_set_detail::hashSimplexID<T> >;
2803 } // 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 subsimplices of a given simplex.

10.18 stringutil.h

[Go to the documentation of this file.](#)

```

1 /*
2 * *****
3 * This file is part of the Colored Abstract Simplicial Complex library.
4 * Copyright (C) 2016-2017
5 * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6 * and Michael Holst
7 *
8 * This library is free software; you can redistribute it and/or
9 * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either

```



```

11  * version 2.1 of the License, or (at your option) any later version.
12  *
13  * This library is distributed in the hope that it will be useful,
14  * but WITHOUT ANY WARRANTY; without even the implied warranty of
15  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16  * Lesser General Public License for more details.
17  *
18  * You should have received a copy of the GNU Lesser General Public
19  * License along with this library; if not, write to the Free Software
20  * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21  *
22  * *****
23  */
24
25 /**
26  * @file stringutil.h
27  * @brief String utilities for CASC.
28  */
29
30 #pragma once
31
32 #include <string>
33
34 namespace casc
35 {
36 /**
37  * @brief Returns a string representation of the vertex subsimplicies
38  * of a given simplex
39  *
40  * @param[in] A Array containing name of a simplex.
41  *
42  * @tparam T Typename KeyType.
43  * @tparam k Dimension of the simplex.
44  *
45  * @return String representation of the object.
46  */
47 template <typename T, std::size_t k>
48 std::string to_string(const std::array<T,k>& A)
49 {
50     if (k==0) {
51         return "{root}";
52     }
53     std::string out;
54     out += "{";
55     for(int i = 0; i + 1 < k; ++i)
56     {
57         out += std::to_string(A[i]) + ",";
58     }
59     if(k > 0)
60     {
61         out += std::to_string(A[k-1]);
62     }
63     out += "}";
64     return out;
65 }
66 } // 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.](#)

```
1 /**
2  * @file typetraits.h
3  * @brief Helper functions for debugging template types.
4  *
5  * This is copied directly from this very helpful post from
6  * <a
7  *   href="https://stackoverflow.com/questions/81870/is-it-possible-to-print-a-variables-type-in-standard-c/20170989#20170989"
8  *
9  * /// @cond hidden
10 #pragma once
11
12 #include <cstdint>
13 #include <stdexcept>
14 #include <cstring>
15 #include <ostream>
16
17 #ifndef _MSC_VER
18 #   if __cplusplus < 201103
19 #       define CONSTEXPR11_TN
20 #       define CONSTEXPR14_TN
21 #       define NOEXCEPT_TN
22 #   elif __cplusplus < 201402
23 #       define CONSTEXPR11_TN constexpr
24 #       define CONSTEXPR14_TN
25 #       define NOEXCEPT_TN noexcept
26 #   else
27 #       define CONSTEXPR11_TN constexpr
28 #       define CONSTEXPR14_TN constexpr
29 #       define NOEXCEPT_TN noexcept
30 #   endif
31 #else // _MSC_VER
32 #   if _MSC_VER < 1900
33 #       define CONSTEXPR11_TN
34 #       define CONSTEXPR14_TN
35 #       define NOEXCEPT_TN
36 #   elif _MSC_VER < 2000
37 #       define CONSTEXPR11_TN constexpr
38 #       define CONSTEXPR14_TN
39 #       define NOEXCEPT_TN noexcept
40 #   else
41 #       define CONSTEXPR11_TN constexpr
42 #       define CONSTEXPR14_TN constexpr
43 #       define NOEXCEPT_TN noexcept
44 #   endif
45 #endif // _MSC_VER
46
47 class static_string
48 {
49     const char* const p_;
50     const std::size_t sz_;
51
52 public:
53     typedef const char* const_iterator;
54
55     template <std::size_t N>
56     CONSTEXPR11_TN static_string(const char(&a) [N]) NOEXCEPT_TN
57         : p_(a)
```

```

58         , sz_(N-1)
59     {}
60
61     CONSTEXPR11_TN static_string(const char* p, std::size_t N) NOEXCEPT_TN
62     : p_(p)
63     , sz_(N)
64     {}
65
66     CONSTEXPR11_TN const char* data() const NOEXCEPT_TN {return p_;}
67     CONSTEXPR11_TN std::size_t size() const NOEXCEPT_TN {return sz_;}
68
69     CONSTEXPR11_TN const_iterator begin() const NOEXCEPT_TN {return p_;}
70     CONSTEXPR11_TN const_iterator end() const NOEXCEPT_TN {return p_ + sz_;}
71
72     CONSTEXPR11_TN char operator[](std::size_t n) const
73     {
74         return n < sz_ ? p_[n] : throw std::out_of_range("static_string");
75     }
76 };
77
78 inline
79 std::ostream&
80 operator<<(std::ostream& os, static_string const& s)
81 {
82     return os.write(s.data(), s.size());
83 }
84 /// @endcond
85
86 /**
87  * @brief      Print the typename of an object at compile time.
88  *
89  * Example usage:
90  * ~~~~~{.cpp}
91  * std::cout << "decltype(i) is " << type_name<decltype(i)>() << '\n';
92  * ~~~~~
93  */
94 template <class T>
95 CONSTEXPR14_TN
96 static_string
97 type_name()
98 {
99     #ifdef __clang__
100         static_string p = __PRETTY_FUNCTION__;
101         return static_string(p.data() + 31, p.size() - 31 - 1);
102     #elif defined(__GNUC__)
103         static_string p = __PRETTY_FUNCTION__;
104         # if __cplusplus < 201402
105             return static_string(p.data() + 36, p.size() - 36 - 1);
106         # else
107             return static_string(p.data() + 46, p.size() - 46 - 1);
108         # endif
109     #elif defined(_MSC_VER)
110         static_string p = __FUNCSIG__;
111         return static_string(p.data() + 38, p.size() - 38 - 7);
112     #endif
113 }

```

10.21 include/casc/util.h File Reference

Metatemplate pack expansion helpers.

```

#include <utility>
#include <array>

```

Data Structures

- struct [util::range< T >](#)
A range object to support range based for loops.
- struct [util::type_holder< Ts >](#)
Queue based data structure to hold list of types.
- struct [util::type_holder< T, Ts... >](#)

- *Partial specialization to allow FIFO access of typenames.*
- struct `util::type_get< k, T >`
Helper to get the kth element from a `type_holder`.
- struct `util::type_get< 0, type_holder< Ts... > >`
Specialization for terminal case.
- struct `util::type_get< k, type_holder< Ts... > >`
Specialization to recursively pop types to get the kth type.
- struct `util::type_map< C, V >`
Map the types in C into $V<T>$.
- struct `util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >`
Maps an integer sequence and typename, F , into outholder.
- struct `util::type_swap< TUPLE, HOLDER_FULL >`
Move a list of types from one container to another.
- struct `util::type_swap< TUPLE, HOLDER< Ts... > >`
Move a list of types from one container to another.
- struct `util::reverse_sequence< Integer, IntegerSequence >`
Reverse an Integer Sequence.
- struct `util::remove_first_val< Integer, IntegerSequence >`
General template for removing the first value from a type holder.
- struct `util::remove_first_val< Integer, InHolder< Integer, I, Is... > >`
Specialization for removing first integer from a sequence of compile time integers.

Namespaces

- namespace `util`
Metatemplate programming utilities namespace.

Functions

- template<typename T >
`range< T > util::make_range (T b, T e)`
Make a range object.
- template<typename T >
`range< T > util::make_range (std::pair< T, T > p)`
Makes a range object.
- template<class Integer, typename IntegerSequence, typename Fn, typename ... Args>
`void util::int_for_each (Fn &&f, Args &&... args)`
Calls a function f . `apply<k>()` for a sequence of integer k 's.

10.22 util.h

[Go to the documentation of this file.](#)

```

1 /*
2  * *****
3  * This file is part of the Colored Abstract Simplicial Complex library.
4  * Copyright (C) 2016-2017
5  * by Christopher Lee, John Moody, Rommie Amaro, J. Andrew McCammon,
6  * and Michael Holst
7  *
8  * This library is free software; you can redistribute it and/or
9  * modify it under the terms of the GNU Lesser General Public
10 * License as published by the Free Software Foundation; either

```

```

11 * version 2.1 of the License, or (at your option) any later version.
12 *
13 * This library is distributed in the hope that it will be useful,
14 * but WITHOUT ANY WARRANTY; without even the implied warranty of
15 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
16 * Lesser General Public License for more details.
17 *
18 * You should have received a copy of the GNU Lesser General Public
19 * License along with this library; if not, write to the Free Software
20 * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA 02110-1301 USA
21 *
22 * *****
23 */
24
25 /**
26  * @file util.h
27  * @brief Metatemplate pack expansion helpers
28  */
29
30 #pragma once
31
32 #include <utility>
33 #include <array>
34
35 /// Metatemplate programming utilities namespace
36 namespace util
37 {
38 /**
39  * @brief      A range object to support range based for loops.
40  *
41  * This is a basic data structure which implements a 'begin()' and 'end()'
42  * functions for range based for looping added in C++11.
43  * See also
44  * <a href="http://en.cppreference.com/w/cpp/language/range-for">range-for</a>.
45  *
46  * @tparam     T      Typename of the iterator
47  */
48 template<typename T> struct range
49 {
50
51     /**
52      * @brief      Construct a range for a container class.
53      *
54      * @param[in]   c      Container class which implements begin() and end().
55      *
56      * @tparam     C      Typename of the container.
57      */
58     template <class C>
59     range(C &c) : _begin(c.begin()), _end(c.end()) {}
60
61     /**
62      * @brief      Construct a range from an iterator.
63      *
64      * @param[in]   b      Beginning iterator
65      * @param[in]   e      End iterator.
66      */
67     range(T b, T e) : _begin(b), _end(e) {}
68
69     /**
70      * @brief      Get the beginning iterator.
71      *
72      * @return      Returns an iterator to the beginning.
73      */
74     T begin() { return _begin; }
75
76     /**
77      * @brief      Get the end iterator.
78      *
79      * @return      Returns an iterator to the end.
80      */
81     T end() { return _end; }
82
83     private:
84         /// Iterator to the beginning.
85         T _begin;
86         /// Iterator to the end.
87         T _end;
88 };
89
90 /**
91  * @brief      Make a range object.
92  *
93  * @param[in]   b      Iterator to the beginning.
94  * @param[in]   e      Iterator to the end.
95  *
96  * @tparam     T      Typename of the iterator.
97  */

```

```

98 * @return      Returns a range of the iterators.
99 */
100 template<typename T> range<T> make_range(T b, T e)
101 {
102     return range<T>(std::move(b), std::move(e));
103 }
104
105 /**
106 * @brief      Makes a range object.
107 * @param[in]  p      A pair containing begin and end iterators.
108 * @tparam    T      Typename of the iterator.
109 * @return     Returns a range of the iterators.
110 */
111 template<typename T> range<T> make_range(std::pair<T, T> p)
112 {
113     return range<T>(std::move(p.first), std::move(p.second));
114 }
115
116 /**
117 * @brief      Queue based data structure to hold list of types.
118 * Types in the type_holder can be accessed by accessing the 'head' type.
119 * Subsequent types are in the 'tail'. See also type_get.
120 * @tparam    Ts      List of typenames
121 */
122 template <typename ... Ts>
123 struct type_holder
124 {
125     /// Length of the list of types
126     static const std::size_t size = sizeof ... (Ts);
127 };
128
129 /**
130 * @brief      Partial specialization to allow FIFO access of typenames.
131 * @tparam    T      The first typename
132 * @tparam    Ts      The following typenames
133 */
134 template <typename T, typename ... Ts>
135 struct type_holder<T, Ts...>
136 {
137     /// The first type
138     using head = T;
139     /// The following types
140     using tail = type_holder<Ts...>;
141     /// Length of the list of types
142     static const std::size_t size = 1 + type_holder<Ts...>::size;
143 };
144
145 /**
146 * @brief      Helper to get the kth element from a type_holder.
147 * This is the empty general template which will be later specialized.
148 * @tparam    k      Integer index of the type to retrieve
149 * @tparam    T      A type_holder queue of typenames
150 */
151 template <std::size_t k, typename T>
152 struct type_get {};
153
154 /**
155 * @brief      Specialization for terminal case.
156 * @tparam    Ts      Following typenames
157 */
158 template <typename ... Ts>
159 struct type_get<0, type_holder<Ts...> >
160 {
161     /// The first type of the type_holder
162     using type = typename type_holder<Ts...>::head;
163 };
164
165 /**
166 * @brief      Specialization to recursively pop types to get the kth type.
167 * @tparam    k      Integral constant of the type to get
168 * @tparam    Ts      List of typenames
169 */
170 template <std::size_t k, typename ... Ts>
171 struct type_get<k, type_holder<Ts...> >
172 {
173     /// Recurse after popping the first type off
174     using type = typename type_get<k-1, typename type_holder<Ts...>::tail>::type;
175 }

```

```

185 };
186
187 /// @cond detail
188 /// Namespace for utility helper functions
189 namespace detail
190 {
191 /**
192  * @brief      Helper to broadcast a list of types into a class.
193  *
194  * @tparam      C      Class to old a list of types
195  * @tparam      V      Class to broadcast the types into
196  * @tparam      Rs      List of resulting types
197  */
198 template <class C, template <typename> class V, typename ... Rs>
199 struct type_map_helper {};
200
201 /**
202  * @brief      Terminal condition: place the mapped types into a tuple
203  *
204  * @tparam      G      Empty tuple
205  * @tparam      V      Type template <class T> class to map into
206  * @tparam      Rs      List of already mapped types
207  */
208 template <template <class ...> class G, template <typename> class V, typename ... Rs>
209 struct type_map_helper<G<>, V, Rs...>
210 {
211     using type = G<Rs...>;
212 };
213
214 /**
215  * @brief      Map types into
216  *
217  * @tparam      G      Tuple of types
218  * @tparam      T      Current type
219  * @tparam      Ts      List of remaining types
220  * @tparam      V      Type template <class T> class to map into
221  * @tparam      Rs      List of already mapped types
222  */
223 template <template < class ...> class G, typename T, typename ... Ts, template <typename> class V,
224         typename ... Rs>
225 struct type_map_helper<G<T, Ts...>, V, Rs...>
226 {
227     using type = typename type_map_helper<G<Ts...>, V, Rs..., V<T> >::type;
228 };
229 // end of namespace detail
230 /// @endcond
231
232 /**
233  * @brief      Map the types in C into 'V<T>'.
234  *
235  * Given a container of types 'C<T1,T2,T3,...>' and template template type
236  * 'V<T>', this function will apply the types in C to 'V<T>'. This produces
237  * 'C<V<T1>, V<T2>, V<T3>, ...>'.
238  *
239  * @tparam      C      Container of compile time types.
240  * @tparam      V      Template template class 'V<T>' to map into.
241  */
242 template <class C, template <typename> class V>
243 struct type_map
244 {
245     /// Tuple of 'C<V<T1>, V<T2>, V<T3>, ...>'
246     using type = typename detail::type_map_helper<C, V>::type;
247 };
248 /// @cond detail
249 namespace detail
250 {
251 /**
252  * @brief      Template for future specialization
253  */
254 template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
255         <IntegerType> class F, typename ... Accumulators>
256 struct int_type_map_helper {};
257
258 /**
259  * @brief      Apply the typenames to the OutHolder
260  *
261  * @tparam      Integer      Integral type
262  * @tparam      OutHolder    Type to ultimately hold the accumulated
263  * @tparam      InHolder     Class of index sequence
264  * @tparam      F            Type to apply index to
265  * @tparam      Accumulator  List of mapped typenames F<I>
266  */
267 template <class Integer, template <class ...> class OutHolder, template <class, Integer...> class
268         InHolder, template <Integer> class F, class ... Accumulator>
269 struct int_type_map_helper<Integer, OutHolder, InHolder<Integer>, F, Accumulator...>
270 {

```

```

269     using type = OutHolder<Accumulator...>;
270 };
271
272 /**
273  * @brief      Iterates across integers and fills accumulator with F<I>
274  *
275  * @tparam      Integer      Integral type
276  * @tparam      OutHolder    Type to ultimately hold the accumulated
277  * @tparam      InHolder     Class of index sequence
278  * @tparam      I            Current integer
279  * @tparam      Is          Next integer(s)
280  * @tparam      F           Type to apply index to
281  * @tparam      Accumulator  List of previously mapped typenames F<I>
282  */
283 template <class Integer, template <class ...> class OutHolder, template <class, Integer...> class
    InHolder, Integer I, Integer... Is, template <Integer> class F, class ... Accumulator>
284 struct int_type_map_helper<Integer, OutHolder, InHolder<Integer, I, Is...>, F, Accumulator...>
285 {
286     using type = typename int_type_map_helper<Integer, OutHolder, InHolder<Integer, Is...>, F,
        Accumulator..., F<I> >::type;
287 };
288 } // end namespace detail
289 /// @endcond
290
291 /**
292  * @brief      Maps an integer sequence and typename, F, into outholder.
293  *
294  * Given an Integer Sequence 'I<0,1,2,3,...>' and template template type 'F<I>',
295  * this function produces 'Out<F<0>, F<1>, F<2>, ...>'.
296  *
297  * @tparam      IntegerType    Typename of an integer type
298  * @tparam      OutHolder      Typename of a holder for types
299  * @tparam      IntegerSequence Integral sequence of types
300  * @tparam      F              Typename of class to be broadcast with integer
301  */
302 template <class IntegerType, template <class ...> class OutHolder, class IntegerSequence, template
    <IntegerType> class F>
303 struct int_type_map
304 {
305     /// Tuple of 'Out<F<0>, F<1>, F<2>, ...>'.
306     using type = typename detail::int_type_map_helper<IntegerType, OutHolder, IntegerSequence, F>::type;
307 };
308
309 /**
310  * @brief      Move a list of types from one container to another.
311  *
312  * @tparam      TUPLE          Empty container
313  * @tparam      HOLDER_FULL    Full container
314  */
315 template <template <class ...> class TUPLE, typename HOLDER_FULL>
316 struct type_swap
317 {};
318
319 /**
320  * @brief      Move a list of types from one container to another.
321  *
322  * @tparam      TUPLE          Empty container
323  * @tparam      HOLDER         Full container
324  * @tparam      Ts            Typenames in full container
325  */
326 template <template <class ...> class TUPLE, template <class ...> class HOLDER, typename ... Ts>
327 struct type_swap<TUPLE, HOLDER<Ts...> >
328 {
329     /// Empty container filled with typenames from full container
330     using type = TUPLE<Ts...>;
331 };
332
333 /// @cond detail
334 namespace detail
335 {
336 /**
337  * @brief      Helper struct to reverse a typed sequence.
338  *
339  * @tparam      Integer      Typename of integer class.
340  * @tparam      IntegerSequence Sequence of integral types.
341  * @tparam      Accumulator  Bucket ot hold types.
342  */
343 template <class Integer, class IntegerSequence, Integer... Accumulator>
344 struct reverse_sequence_helper {};
345
346 /**
347  * @brief      Terminal case of typed sequence reversal.
348  *
349  * @tparam      Integer      Typename of an integer class
350  * @tparam      InHolder     Template template type holder
351  * @tparam      Accumulator  List of reverse ordered typenames.
352  */

```



```

353 template <class Integer,
354           template<class, Integer...> class InHolder,
355           Integer... Accumulator>
356 struct reverse_sequence_helper<Integer, InHolder<Integer>, Accumulator...>
357 {
358     /// Reversed sequence of types.
359     using type = InHolder<Integer, Accumulator...>;
360 };
361
362 /**
363  * @brief      Helper struct to reverse a typed sequence.
364  *
365  * @tparam      Integer      Typename of integer class.
366  * @tparam      InHolder     Type holder.
367  * @tparam      I            First type in InHolder.
368  * @tparam      Is          The following types in InHolder.
369  * @tparam      Accumulator  List of reversed typenames.
370  */
371 template <class Integer,
372           template<class, Integer...> class InHolder,
373           Integer I, Integer... Is,
374           Integer... Accumulator>
375 struct reverse_sequence_helper<Integer, InHolder<Integer, I, Is...>, Accumulator...>
376 {
377     // Push the first type into the Accumulator and recurse.
378     /// Reversed sequence of types.
379     using type = typename reverse_sequence_helper<Integer,
380                                                  InHolder<Integer, Is...>, I, Accumulator...>::type;
381 };
382 } // end namespace detail
383 /// @endcond
384
385 /**
386  * @brief      Reverse an Integer Sequence
387  *
388  * @tparam      Integer      Typename of an integer class.
389  * @tparam      IntegerSequence  Sequence of compile-time integers.
390  */
391 template <class Integer, class IntegerSequence>
392 struct reverse_sequence
393 {
394     /// Reversed sequence of types.
395     using type = typename detail::reverse_sequence_helper<Integer, IntegerSequence>::type;
396 };
397
398
399 /**
400  * @brief      General template for removing the first value from a type holder.
401  *
402  * @tparam      Integer      Typename of integer.
403  * @tparam      IntegerSequence  Sequence of compile time integers.
404  */
405 template <class Integer, class IntegerSequence>
406 struct remove_first_val {};
407
408 /**
409  * @brief      Specialization for removing first integer from a sequence of
410  *              compile time integers.
411  *
412  * @tparam      Integer      Typename of integer type.
413  * @tparam      InHolder     Type holder of integer sequence.
414  * @tparam      I            The first integer
415  * @tparam      Is          Remaining integers
416  */
417 template <class Integer,
418           template<class, Integer...> class InHolder,
419           Integer I, Integer... Is>
420 struct remove_first_val<Integer, InHolder<Integer, I, Is...> >
421 {
422     /// Type holder with first value removed.
423     using type = InHolder<Integer, Is...>;
424 };
425
426 /// @cond detail
427 namespace detail
428 {
429 /**
430  * @brief      Template type for future specialization
431  *
432  */
433 template <typename Integer, typename IntegerSequence, typename Fn, typename ... Args>
434 struct int_for_each_helper {};
435
436 /**
437  * @brief      Terminal Case
438  *
439  * @tparam      Integer      { description }

```

```

440 * @tparam InHolder { description }
441 * @tparam I { description }
442 * @tparam Fn { description }
443 * @tparam Args { description }
444 */
445 template <class Integer, template <class, Integer...> class InHolder,
446         Integer I, typename Fn, typename ... Args>
447 struct int_for_each_helper<Integer, InHolder<Integer, I>, Fn, Args...>
448 {
449     static void apply(Fn &&f, Args && ... args)
450     {
451         f.template apply<I>(std::forward<Args>(args) ...);
452     }
453 };
454
455 /**
456 * @brief Recurse through the integer series
457 *
458 * @tparam Integer { description }
459 * @tparam InHolder { description }
460 * @tparam I { description }
461 * @tparam Is { description }
462 * @tparam Fn { description }
463 * @tparam Args { description }
464 */
465 template <class Integer, template <class, Integer...> class InHolder,
466         Integer I, Integer... Is, typename Fn, typename ... Args>
467 struct int_for_each_helper<Integer, InHolder<Integer, I, Is...>, Fn, Args...>
468 {
469     static void apply(Fn &&f, Args && ... args)
470     {
471         f.template apply<I>(std::forward<Args>(args) ...);
472         int_for_each_helper<Integer, InHolder<Integer, Is...>, Fn, Args...>::apply(
473             std::forward<Fn>(f),
474             std::forward<Args>(args) ...);
475     }
476 };
477 } // end namespace detail
478 /// @endcond
479
480 /**
481 * @brief Calls a function 'f.apply<k>()' for a sequence of integer k's
482 *
483 * @param[in] args Arguments to f
484 * @param[in] f Functor with 'apply<k>()' method
485 *
486 * @tparam Integer Integer type
487 * @tparam IntegerSequence Sequence of integers to iterate
488 * @tparam Fn Typename of functor f
489 * @tparam Args Typenames of the arguments
490 */
491 template <class Integer, typename IntegerSequence, typename Fn, typename ... Args>
492 void int_for_each(Fn &&f, Args && ... args)
493 {
494     detail::int_for_each_helper<Integer, IntegerSequence, Fn, Args...>::apply(std::forward<Fn>(f),
495                                     std::forward<Args>(args) ...);
496 }
497 } // End of namespace util

```

Index

- `~simplicial_complex`
 - `casc::simplicial_complex< traits >`, 79
- `AbstractSimplicialComplex`
 - `casc`, 26
- `add_vertex`
 - `casc::simplicial_complex< traits >`, 79
- `begin`
 - `casc::SimplexSet< Complex >`, 69
 - `util::range< T >`, 58
- `casc`, 23
 - `AbstractSimplicialComplex`, 26
 - `check_orientation`, 26
 - `clear_orientation`, 27
 - `compute_orientation`, 27
 - `decimate`, 27
 - `decimateBackHalf`, 28
 - `decimateFirstHalf`, 28
 - `edge_up`, 29
 - `get`, 29, 30
 - `getClosure`, 30, 31
 - `getLink`, 31
 - `getStar`, 32
 - `init_orientation`, 33
 - `kneighbors`, 33, 34
 - `kneighbors_up`, 34, 35
 - `neighbors`, 35, 36
 - `neighbors_up`, 36, 37
 - `operator!=`, 37
 - `operator==`, 38
 - `perform_insertion`, 38
 - `perform_removal`, 38
 - `run_user_callback`, 39
 - `set_difference`, 39
 - `set_intersection`, 40
 - `set_union`, 40
 - `to_string`, 41
 - `visit_BFS_down`, 41
 - `visit_BFS_up`, 41
 - `writeDOT`, 42
- `casc::Orientable`, 56
- `casc::SimplexMap< Complex >`, 65
 - `get`, 66, 67
 - `operator<<`, 67
- `casc::SimplexSet< Complex >`, 68
 - `begin`, 69
 - `cbegin`, 70
 - `cend`, 70
 - `empty`, 70
 - `end`, 71
 - `erase`, 71
 - `find`, 72
 - `insert`, 73
 - `operator<<`, 74
 - `size`, 73
- `casc::simplicial_complex< traits >`, 74
 - `~simplicial_complex`, 79
 - `add_vertex`, 79
 - `down`, 80, 81
 - `EdgeData`, 78
 - `EdgeID`, 98
 - `eq`, 81
 - `exists`, 82
 - `get_cover`, 82, 83
 - `get_cover_insert`, 83
 - `get_edge_down`, 84
 - `get_edge_up`, 85
 - `get_level`, 86
 - `get_level_id`, 86, 87
 - `get_name`, 87, 88
 - `get_simplex_down`, 88, 89
 - `get_simplex_up`, 89–91
 - `insert`, 91, 92
 - `leq`, 93
 - `lt`, 93
 - `nearBoundary`, 94
 - `NodeData`, 79
 - `onBoundary`, 94
 - `remove`, 95
 - `SimplexID`, 98
 - `size`, 96
 - `up`, 96, 97
- `casc::simplicial_complex< traits >::EdgeID< k >`, 50
 - `down`, 52
 - `EdgeID`, 51, 52
 - `up`, 52
- `casc::simplicial_complex< traits >::SimplexID< k >`, 60
 - `cover`, 62
 - `cover_insert`, 63
 - `get_simplex_up`, 63, 64
 - `indices`, 64
 - `operator<<`, 65
 - `SimplexID`, 62
- `cbegin`
 - `casc::SimplexSet< Complex >`, 70
- `cend`
 - `casc::SimplexSet< Complex >`, 70

check_orientation
 casc, 26
 clear_orientation
 casc, 27
 compute_orientation
 casc, 27
 cover
 casc::simplicial_complex< traits >::SimplexID< k
 >, 62
 cover_insert
 casc::simplicial_complex< traits >::SimplexID< k
 >, 63
 decimate
 casc, 27
 decimateBackHalf
 casc, 28
 decimateFirstHalf
 casc, 28
 down
 casc::simplicial_complex< traits >, 80, 81
 casc::simplicial_complex< traits >::EdgeID< k >,
 52
 edge_up
 casc, 29
 EdgeData
 casc::simplicial_complex< traits >, 78
 EdgeID
 casc::simplicial_complex< traits >, 98
 casc::simplicial_complex< traits >::EdgeID< k >,
 51, 52
 empty
 casc::SimplexSet< Complex >, 70
 end
 casc::SimplexSet< Complex >, 71
 util::range< T >, 58
 eq
 casc::simplicial_complex< traits >, 81
 erase
 casc::SimplexSet< Complex >, 71
 exists
 casc::simplicial_complex< traits >, 82
 find
 casc::SimplexSet< Complex >, 72
 get
 casc, 29, 30
 casc::SimplexMap< Complex >, 66, 67
 get_cover
 casc::simplicial_complex< traits >, 82, 83
 get_cover_insert
 casc::simplicial_complex< traits >, 83
 get_edge_down
 casc::simplicial_complex< traits >, 84
 get_edge_up
 casc::simplicial_complex< traits >, 85
 get_level
 casc::simplicial_complex< traits >, 86
 get_level_id
 casc::simplicial_complex< traits >, 86, 87
 get_name
 casc::simplicial_complex< traits >, 87, 88
 get_simplex_down
 casc::simplicial_complex< traits >, 88, 89
 get_simplex_up
 casc::simplicial_complex< traits >, 89–91
 casc::simplicial_complex< traits >::SimplexID< k
 >, 63, 64
 getClosure
 casc, 30, 31
 getLink
 casc, 31
 getStar
 casc, 32
 include/casc/CASCFUNCTIONS.h, 105, 106
 include/casc/CASCTraversals.h, 111, 113
 include/casc/decimate.h, 122, 123
 include/casc/index_tracker.h, 130, 132
 include/casc/Orientable.h, 143, 144
 include/casc/SimplexMap.h, 147, 148
 include/casc/SimplexSet.h, 150, 151
 include/casc/SimplicialComplex.h, 158, 160
 include/casc/stringutil.h, 192
 include/casc/typetraits.h, 193, 194
 include/casc/util.h, 195, 196
 index_tracker, 42
 index_tracker::index_tracker< _T, _d >, 54
 index_tracker::index_tracker< _T, _d >, 53
 index_tracker, 54
 index_tracker::index_tracker_detail, 43
 index_tracker::index_tracker_detail::BTreeNode< _T, _d
 >, 49
 index_tracker::index_tracker_detail::Interval< T >, 55
 operator=, 55
 indices
 casc::simplicial_complex< traits >::SimplexID< k
 >, 64
 init_orientation
 casc, 33
 insert
 casc::SimplexSet< Complex >, 73
 casc::simplicial_complex< traits >, 91, 92
 int_for_each
 util, 45
 neighbors
 casc, 33, 34
 neighbors_up
 casc, 34, 35
 leq
 casc::simplicial_complex< traits >, 93
 lt
 casc::simplicial_complex< traits >, 93

- make_range
 - util, [46](#)
- nearBoundary
 - casc::simplicial_complex< traits >, [94](#)
- neighbors
 - casc, [35](#), [36](#)
- neighbors_up
 - casc, [36](#), [37](#)
- NodeData
 - casc::simplicial_complex< traits >, [79](#)
- onBoundary
 - casc::simplicial_complex< traits >, [94](#)
- operator!=
 - casc, [37](#)
- operator<<
 - casc::SimplexMap< Complex >, [67](#)
 - casc::SimplexSet< Complex >, [74](#)
 - casc::simplicial_complex< traits >::SimplexID< k >, [65](#)
- operator=
 - index_tracker::index_tracker_detail::Interval< T >, [55](#)
- operator==
 - casc, [38](#)
- perform_insertion
 - casc, [38](#)
- perform_removal
 - casc, [38](#)
- range
 - util::range< T >, [57](#)
- remove
 - casc::simplicial_complex< traits >, [95](#)
- run_user_callback
 - casc, [39](#)
- set_difference
 - casc, [39](#)
- set_intersection
 - casc, [40](#)
- set_union
 - casc, [40](#)
- SimplexID
 - casc::simplicial_complex< traits >, [98](#)
 - casc::simplicial_complex< traits >::SimplexID< k >, [62](#)
- size
 - casc::SimplexSet< Complex >, [73](#)
 - casc::simplicial_complex< traits >, [96](#)
- to_string
 - casc, [41](#)
- type_name
 - typetraits.h, [194](#)
- typetraits.h
 - type_name, [194](#)
- up
 - casc::simplicial_complex< traits >, [96](#), [97](#)
 - casc::simplicial_complex< traits >::EdgeID< k >, [52](#)
- util, [44](#)
 - int_for_each, [45](#)
 - make_range, [46](#)
- util::int_type_map< IntegerType, OutHolder, IntegerSequence, F >, [54](#)
- util::range< T >, [56](#)
 - begin, [58](#)
 - end, [58](#)
 - range, [57](#)
- util::remove_first_val< Integer, InHolder< Integer, I, Is... > >, [59](#)
- util::remove_first_val< Integer, IntegerSequence >, [58](#)
- util::reverse_sequence< Integer, IntegerSequence >, [59](#)
- util::type_get< 0, type_holder< Ts... > >, [99](#)
- util::type_get< k, T >, [98](#)
- util::type_get< k, type_holder< Ts... > >, [99](#)
- util::type_holder< T, Ts... >, [100](#)
- util::type_holder< Ts >, [100](#)
- util::type_map< C, V >, [101](#)
- util::type_swap< TUPLE, HOLDER< Ts... > >, [102](#)
- util::type_swap< TUPLE, HOLDER_FULL >, [102](#)
- visit_BFS_down
 - casc, [41](#)
- visit_BFS_up
 - casc, [41](#)
- writeDOT
 - casc, [42](#)