

# libsequence tutorial

Kevin Thornton

April 13, 2006

## Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	What this document isn't . . . . .	4
1.2	What you need to already know . . . . .	4
1.3	Compiler requirements . . . . .	5
1.4	Compiling and linking . . . . .	5
1.5	Physical Structure of the Library . . . . .	6
1.6	namespace . . . . .	6
1.7	Exception Handling . . . . .	6
1.8	Further documentation . . . . .	7
<b>2</b>	<b>Reading sequences</b>	<b>8</b>
2.1	Inheritance hierarchy . . . . .	8
2.2	Typecast to <code>std::string</code> . . . . .	8
2.3	Example: reverse and complement a file of FASTA sequences .	8
2.4	Example: get a list of sequence names from a file of FASTA sequences . . . . .	9
2.5	Other useful sequence features . . . . .	10
2.5.1	Typecast to <code>std::string</code> . . . . .	10
2.6	Is it a valid sequence? . . . . .	11
2.7	Translating sequences . . . . .	12
2.8	Generating Codon Tables . . . . .	13
<b>3</b>	<b>Handling Alignments</b>	<b>15</b>
3.1	Example: reading an alignment of FASTA sequences . . . . .	15
3.2	Example: convert a clustalw output file to FASTA . . . . .	16

3.3	A digression: template instantiation in <code>libsequence</code> . . . . .	18
<b>4</b>	<b>Basics of Polymorphism Analysis</b>	<b>20</b>
4.1	Inheritance hierarchy . . . . .	20
4.2	Acceptable characters and assumptions about phase . . . . .	20
4.3	Accessing data on a <code>PolyTable</code> . . . . .	21
4.4	Example: calculate Tajima's D from an alignment of DNA sequence data . . . . .	22
4.4.1	Comments on outgroup sequences . . . . .	27
4.4.2	An important caveat . . . . .	27
4.5	Example: calculate the frequency spectrum from coalescent simulation data . . . . .	28
4.6	Example: obtain the distribution of Tajima's D from coales- cent simulation . . . . .	30
4.7	Counting the states at a site in a polymorphism table . . . . .	31
4.7.1	A more general counter . . . . .	33
4.8	One last class: <code>SimpleSNP</code> . . . . .	33
4.9	Other member functions of polymorphism table classes . . . . .	34
4.9.1	Obtaining the "raw" data . . . . .	34
4.9.2	Accessing the positions of segregating sites . . . . .	35
4.9.3	Removing sites with more than two alleles . . . . .	35
4.9.4	Removing missing data from a polymorphism table . . . . .	35
4.9.5	Removing sites below a certain frequency . . . . .	36
4.9.6	Converting the data into a binary format (and using objects derived from <code>Sequence::PolyTable</code> objects with C code) . . . . .	36
4.9.7	Assigning data to a <code>PolyTable</code> . . . . .	37
4.10	Iterators for polymorphism table classes . . . . .	37
4.10.1	A digression—how to represent SNP data? . . . . .	37
4.11	Iterators to site positions . . . . .	38
4.12	Iterators to individuals . . . . .	38
4.13	Iterators to segregating sites . . . . .	38
4.13.1	Why this can be an expensive iterator . . . . .	39
4.13.2	Don't invalidate your pointers! . . . . .	39
4.14	"Rotating" a SNP table . . . . .	41
4.15	The header <code>Sequence/PolyTableFunctions.hpp</code> . . . . .	43

<b>5</b>	<b>Counting</b>	<b>44</b>
5.1	Sequence/CountingOperators.hpp . . . . .	44

# 1 Introduction

This document provides a tutorial for `libsequence`, a C++ API for DNA sequence analysis (with an emphasis on molecular population genetics and molecular evolution). In this introductory section, we will cover some basic ideas. This document does not cover installation of the library itself. This document assumes a Unix or similar system. Linux and Apple's OS X (<http://www.apple.com>) work well, and `libsequence` is well-tested on those platforms.

## 1.1 What this document isn't

This document is not a tutorial on how to extend `libsequence` by deriving custom classes from the existing code base. While such things are possible, they first require a rather thorough reading of the developer's reference manual in the sections you are interested in extending before jumping in.

## 1.2 What you need to already know

1. The jargon of molecular population genetics. This document is largely a tutorial for people writing code for genomics and evolutionary genetics. I assume that you're reading this because you're in a line of work where this is useful to you, so I will refrain from discussing the underlying biology unless absolutely necessary (i.e. if it illustrates why or how a piece of code does what it does).
2. Be comfortable operating in a Unix environment
3. How to compile source code (`g++ -o my_prog my_prog.cc`)
4. C++ (and its C subset). I recommend Stroustrup's description of the language [7] and Scott Meyer's trilogy [4, 5, 6], which focuses more on *design* in C++ and on the proper use of C++ idioms than on describing C++ syntax<sup>1</sup>. The following C++ features are important for reading this document:

---

<sup>1</sup>Beginners with C++ who haven't read Meyer's book should do so as soon as possible. The first two books ([4, 5]) are particularly useful, being two of the kind of book that makes the light bulb go on in your head and rewrite a lot of code.

- (a) The functions that C++ implicitly generates for classes and their behavior (especially implicit copy constructors), and how the keyword `explicit` modifies this.
  - (b) inheritance hierarchies
  - (c) iterators
  - (d) C/C++ compatibility (Item 34 of [5])
  - (e) function objects ("functors", Items 38-42 of [6])
  - (f) the standard algorithms and how they interact with iterators and functors
5. C++ templates. I list this separately so that I can recommend Vandevorde and Josuttis "C++ Templates" [9]. This book covers everything you need to know about templates.
6. Reading developer's reference manuals. `libsequence` can do a lot for you, and a tutorial is not a full description of an API. Also, I do not guarantee that the discussion of any class in this document will discuss all of its member functions, so perusing the reference manual (<http://www.molpopgen.org/software/libsequence/doc/html>) is a good idea. Also, the online docs contain loads of code snippets that serve as extra examples.

### 1.3 Compiler requirements

`gcc` is the preferred compiler system (<http://gcc.gnu.org>). The `gcc 3.x` series is required, and the older 2.9x series is not officially supported. `libsequence` is a bit demanding in some places about ANSI C++ compliance, so a good compiler is needed.

### 1.4 Compiling and linking

To compile a program linking to `libsequence`, just add `-lsequence` to your compile command:

```
g++ -o foo foo.cc -lsequence
```

## 1.5 Physical Structure of the Library

As of libsequence 1.3.2, all library headers are in the subdirectory `Sequence`, so the correct include directive for a header `header.hpp` is:

```
#include <Sequence/header.hpp>
```

## 1.6 namespace

The primary namespace in `libsequence` is `Sequence`. There are "sub" namespaces as well, and we'll cover them as necessary. Remember that there are two ways to import a name from a namespace into scope:

1. `using namespace Sequence;`
2. `using Sequence::foo;`

The first method brings *all* symbols from the namespace into scope in the current translation unit (read: the current source file). More precisely, all symbols for which declarations have been made accessible in the current translation unit via `#include` directives are brought into scope and can be operated on without explicit reference to the namespace. The second only brings the specific symbol (`Sequence::foo`) into scope (provided, of course, that the declaration of the symbol is accessible).

## 1.7 Exception Handling

The class `Sequence::SeqException` is the base exception class in `libsequence`. The class is declared in `<Sequence/SeqExceptions.hpp>`. Its constructor takes a `const char *` describing the error message. `operator<<` is defined for this class, so that messages can be printed to streams. Currently, only one class, `Sequence::badFormat` is derived from `Sequence::SeqException`, and it is thrown when input operations encounter badly-formatted data. For all other exceptional situations, `Sequence::SeqException` is thrown.

Beginning with the 1.3.x releases of the library, exception specifications have been creeping their way into the library routines. If you don't know what exception specifications are, Meyers has a good discussion of them in [5]. I'm rather confident that functions with such specifications behave properly, as I've yet to come across an instance of `unexpected` occurring.

## 1.8 Further documentation

`libsequence` has a rather large reference manual listing all functions, classes, operators, etc., that are provided by the library. The manual is generated directly from the source code using the `doxygen` tool (<http://www.doxygen.org>). A fully cross-referenced html version of the manual can be found at <http://www.molpopgen.org/software/libsequence/doc/html/>, and a downloadable pdf at <http://www.molpopgen.org/software/libsequence/doc/refman.pdf>.

## 2 Reading sequences

### 2.1 Inheritance hierarchy

The base class for sequences is the abstract class `Sequence::Seq`, which defines the interface for all sequence types. Currently, only `Sequence::Fasta` derives from it, which is ok since FASTA formatted data is probably the most common sort. `Sequence::Seq` mandates that `operator>>` and `operator<<` be defined in derived classes, so the following code is valid:

```
Sequence::Fasta x;
std::cin >> x;
std::cout << x << std::endl;
```

`Sequence::Seq` publicly inherits from `std::pair<std::string, std::string>`.

The label/name of the sequence can be access via the public member `first`, and the sequence itself is `second` (look up `std::pair` in your C++ book if you're not familiar with it).

### 2.2 Typecast to `std::string`

Class `Sequence::Seq` provides `operator std::string() const`, which allows typecast of sequences to `std::string`. Note that this operator returns the sequence data, not the sequence label! This operator is provided for compatibility with functions that act on strings. You should also be aware that this operator allows *implicit* typecast of sequences to strings.

### 2.3 Example: reverse and complement a file of FASTA sequences

Let's do our first complete example. We'll read a file of sequences in FASTA format from `stdin`, reverse and complement each sequence, then print the sequences back to `stdout`. This program will make use of `operator>>`, `operator<<`, and `Sequence::Seq::Revcom()`, which is the member function that actually reverses and complements each sequence.

```
#include <Sequence/Fasta.hpp> //declaration of Sequence::Fasta
#include <iostream>
```

```
int main(int argc, char **argv)
{
```

```

Sequence::Fasta fseq; //declare a variable of type
                    //Sequence::Fasta
while( ! cin.eof() ) //read through the file
{
    std::cin >> fseq;
    fseq.Revcom(); // "Revcom" the sequence
    std::cout << fseq << std::endl; //write results to stdout
}
return 0;
}

```

Note that by calling the `Revcom()` member function, the sequence object itself is changed. This is so that the amount of copying (and the number of objects stored) is minimized. Also, in practice, there are few occasions where one needs both a sequence and its reverse-and-complement (the one notable exception would be pattern-matching). Further, when sequence objects are passed to functions, they will generally be passed as `const`, so one would need to make a copy anyways (either by having the `Revcom()` routine return a new object, as is the case in `bioperl`, or make a copy and then rev/com it). To keep the original while making sure that extra copies persist for as short a time as possible, use a pointer and generate a temporary :

```

Fasta *copy = new Fasta(fseq); //use copy constructor
copy->Revcom();
//need to de-reference pointer,
//or else you're just printing out the address
cout << *copy << endl;
delete copy;

```

## 2.4 Example: get a list of sequence names from a file of FASTA sequences

To do this, we make use of `Sequence::Seq::GetName()`, which returns a `std::string` representing the FASTA header. There is also `Sequence::Seq::GetSeq()` to get a `std::string` representing the sequence itself.

```

#include <Sequence/Fasta.hpp> //declaration of Sequence::Fasta
#include <iostream>

```

```

int main(int argc, char **argv)
{
    Sequence::Fasta fseq; //declare a variable of type
                          //Sequence::Fasta
    while( ! cin.eof()) //read through the file
        {
            std::cin >> fseq;
            std::cout << fseq.GetName() << std::endl;
        }
    return 0;
}

```

## 2.5 Other useful sequence features

In addition to the member functions described above, `Sequence::Seq` defines other useful routines. Several of these routines are just wrappers for their equivalents in `std::string` and will be familiar to C++ programmers. The following list describes these functions. Namespace qualifications are removed for space reasons, so `Seq` refers to `Sequence::Seq` and `string` refers to `std::string`. All of these functions are also overloaded with their `const` equivalents where that would be expected. The `Sequence::Seq` analogs to `std::string` member functions are:

<code>unsigned Seq::length()</code>	return the length of the sequence
<code>string Seq::substr(unsigned beg,unsigned end)</code>	return a string from indexes beg to end
<code>string Seq::substr(unsigned beg)</code>	return a string from beg to the end of the data
<code>string::iterator Seq::begin()</code>	return an iterator to the beginning of the data
<code>string::iterator Seq::end()</code>	return an iterator to the end of the data
<code>const char * Seq::c_str()</code>	return the data as a const char *
<code>char &amp; operator[unsigned j]</code>	return the $j_{th}$ character in the sequence

### 2.5.1 Typecast to `std::string`

Starting with the 1.3.x series, objects inheriting from `Sequence::Seq` can be implicitly typecast to `std::string` via `Sequence::operator std::string()`. This is useful for compatibility with other code where the operations may take a `std::string` as an argument. For compatibility with C code, we noted in 2.5 that there is a member function `Seq::c_str()` that returns the data as a `char *`.

## 2.6 Is it a valid sequence?

Not all data sources may contain the characters you expect. `libsequence` provides a header, `<Sequence/SeqUtilities.hpp>` which provides some functions to allow you to check if a sequence object contains the expected characters. These sorts of operations are often done with regular expressions (regexes), for example by testing for the presence of invalid characters (i.e. the *complement* of the set of valid characters). The routines in this header use the BOOST regex library (<http://www.boost.org>). Note that although `libsequence` has compilation dependencies on the BOOST headers, you may need to do additional work to get the boost regex library working, and you'll need to read the instructions at [www.boost.org](http://www.boost.org) to learn how to compile and install boost libs.

`<Sequence/SeqUtilities.hpp>` defines three regexes in namespace `Sequence`. The regexes are for the complement of the expected characters, and their names are self-explanatory:

```
const char *basic_dna_alphabet = "[^AGTCN\\-]";
const char *full_dna_alphabet = "[^AGCTNXMRWSKVHDB\\-]";
const char *pep_alphabet = "[^ARNDBCQEZGHILKMFPTWYV\\-]";
```

The above regexes are arguments to the following function:

```
template<class Iter>
bool Sequence::validSeq(Iter beg, Iter end,
    const char *pattern = Sequence::basic_dna_alphabet,
    const bool  icase = true)
```

The final boolean argument, `icase`, results in case-insensitive matching when true, case sensitive when false. Here is an example, but remember it won't compile without the BOOST libraries installed:

```
#include <Sequence/Fasta.hpp>
#include <Sequence/SeqUtilities.hpp>
#include <fstream>
#include <iostream>

int main(int argc, char **argv)
{
    std::ifstream in(argv[1]);
```

```

Sequence::Fasta seq;
while ( !in.eof() )
{
    in >> seq;
    std::cout << Sequence::validSeq(seq.begin(),seq.end())
    << '\t'
    << Sequence::validSeq(seq.begin(),seq.end()),
        Sequence::full_dna_alphabet)
    <<'\n';
}
}

```

## 2.7 Translating sequences

`libsequence` provides a function, `Sequence::Translate` to translate DNA sequences to protein. Currently, only the universal genetic code is supported. The prototype is in the header `<Sequence/Translate.hpp>`:

```

std::string Translate(std::string::const_iterator beg,
    std::string::const_iterator end,
    Sequence::SeqEnums::GeneticCodes genetic_code
    = Sequence::SeqEnums::UNIVERSAL);

```

There are several things to note. First, the return type is `std::string`, and contains the amino acid sequence. The argument types are `const_iterators` for `std::string`, meaning that this function is a *range operation* similar to many C++ functions. Note that the iterator types of `Sequence::Seq` are equivalent to those of `std::string`, so this function is compatible with both types. The third argument can be ignored because of its default value, which tells the function to translate using the universal code. The third argument is there in case more genetic codes are supported in the future. Let's translate a Fasta sequence:

```

#include <Sequence/Fasta.hpp>
#include <Sequence/Translate2.hpp>
Sequence::Fasta x;
cin >> x;
cout << Sequence::Translate(x.begin(),x.end()) << endl;

```

If the sequence length is not a multiple of 3, an X character will represent the translation of the partial codon at the end. An X will also represent the translation of codons with missing data if the missing data makes the codon ambiguous. Remember that since iterators are really pointers (at least as far as this example is concerned), they can be added and subtracted from, meaning that we can translate only bits of sequences as we see fit.

## 2.8 Generating Codon Tables

While on the topic of translating sequence, we should also discuss getting a list of codons present in a sequence. The relevant header is `Sequence/CodonTable.hpp`, which contains the following prototypes:

```
Sequence::CodonUsageTable
    makeCodonUsageTable(const Sequence::Seq * seq);
Sequence::CodonUsageTable
    makeCodonUsageTable(const std::string & seq);
```

Yes, that's an inconsistency between the two prototypes. The reason is that the first prototype is a more common declaration of a function that takes polymorphic types (anything derived from `Sequence::Seq` in this case) as arguments. Such functions are usually declared as taking pointers to base classes as arguments. It is not possible to derive classes from `std::string` as its destructor is non-virtual, and the convention in `libsequence` is to pass `const references` to functions that don't modify data (this is adhered to with rather few exceptions).

As of `libsequence` version 1.3.2, the above are the only two declarations available, but in the future, the following will exist:

```
Sequence::CodonUsageTable (std::string::const_iterator beg,
    std::string::const_iterator end);
```

This last prototype will be compatible with both `std::string` and with `Sequence::Seq` and provide niceties such as being able to do the calculation over portions of sequences (such as the coding portions of sequences that contain both coding and non-coding regions).

The return type, `Sequence::CodonUsageTable` is simply a typedef for `std::vector< std::pair<std::string,unsigned> >`. The first member

of each pair is the codon, the second is the number of occurrences of that codon in the sequence. Since the return type is a typedef for `std` types, and those types are described in your favorite C++ book [7], we're done here.

## 3 Handling Alignments

In Section 2, we went over how to read in one sequence at a time. However, we often want to deal with aligned data for analysis, and to be able to assert that the data that we have read in are in fact aligned. The handling of aligned data in `libsequence` is complicated by two factors. The first is that one must distinguish reading in aligned sequences from a flat file (i.e. a file of FASTA sequences) from reading in the output of an alignment program (i.e. an `aln` file produced by ClustalW). The second factor is that routines to handle alignments are implemented as templates in `namespace Sequence::Alignment`. Templates are only a complication insofar as they bring up loads of technicalities: When must I specify the type argument to a template? What instantiation model does my compiler support? Consult [9] for a discussion of these issues.

### 3.1 Example: reading an alignment of FASTA sequences

In this example, we will read from a file of FASTA sequences and ensure that they are aligned. The check for alignment is simply to make sure that all sequences are of the same length, i.e. we make no comments on the quality of the alignment. In this example, be sure to note that automatic type resolution during the template instantiation process will allow us to omit the template type argument when using functions in `namespace Sequence::Alignment`, thus simplifying our syntax.

```
#include <vector>
#include <iostream>
#include <Sequence/Fasta.hpp>
#include <Sequence/Alignment.hpp> //defines namespace Sequence::Alignment

//using declarations make it more explicit where
//we get our routines from
using std::vector;
using std::cerr;
using std::endl;
using Sequence::Fasta;
using Sequence::Alignment::GetData;
using Sequence::Alignment::IsAlignment;
```

```

int main(int argc, char **argv)
{
    vector<Fasta> data;
    char *infile = argv[1];
    GetData(data,infile);
    if( ! IsAlignment(data) )
    {
        cerr << "file not aligned!" << endl;
    }
    else
    {
        cerr << "file aligned" << endl;
    }
    return 0;
}

```

## 3.2 Example: convert a clustalw output file to FASTA

The example in this section will be to read in an alignment file generated by clustalw and output a file of the data in FASTA format. This example will introduce several new concepts:

1. Exception handling using both class `Sequence::SeqException` and `std::exception`
2. The interoperability of types from namespace `Sequence` and the Standard Template Library (STL).
3. Range operations using `Sequence::ClustalW<Fasta>`.

```

#include <iostream>
#include <iterator>
#include <fstream>
#include <algorithm>
#include <stdexcept>
#include <Sequence/Fasta.hpp>
#include <Sequence/Clustalw.hpp>
#include <Sequence/SeqExceptions.hpp>

using std::ostream_iterator;

```

```

using std::ifstream;
using std::ofstream;
using std::cerr;
using std::copy;
using std::exception;
using Sequence::Fasta;
using Sequence::ClustalW;
using Sequence::SeqException;

int main(int argc, char **argv)
{
    char *infile = argv[1];
    char *outfile = argv[2];
    //Sequence::ClustalW is a template class,
    //so we need to specify the type (Sequence::Fasta here)
    ClustalW<Fasta> alignment;
    ifstream in(infile);
    try
    {
        //read the alignment from the file
        in >> alignment;
    }
    catch (SeqException &e)
    {
        //Sequence::SeqException (or a derived class)
        //may be thrown if there is a problem with the input
        cerr << e << endl;
        exit(10);
    }
    catch (exception &e)
    {
        //handle std exceptions
        //hopefully we'll never need to...
        cerr << e.what() << endl;
        exit(10);
    }

    //output FASTA sequences to file
    //We do this by copying to a std::ostream_iterator
    //There are 2 things to note here:

```

```

//1.) How Sequence::Fasta fits right into
//an STL template class
//2.) The use of ClustalW::begin() and ::end() to perform
//a stereotypical C++ range operation
ofstream out(outfile);
copy(alignment.begin(),alignment.end(),
      ostream_iterator<Fasta>(out,"\n"));
return 0;
}

```

Note how short the above code is. All the I/O is handled in 4 lines of code, 2 of which are devoted to opening the `std::ifstream` and `std::ofstream`. A lot of headers get included (relatively), but that's a small price to pay considering that we avoided all need to write explicit loops and we get error handling to boot. The program also uses most of the important design aspects of C++ (objects, exception handling, iterators, and templates). And the code would be even shorter if we just brought the namespaces `std` and `Sequence` directly into scope *en lieu* of the individual `using` directives (see subsection 1.6, page 6). In case you're not familiar with it, the use of `std::ostream_iterator<T>` is discussed in Item 49 of [6]. A functionally equivalent alternative to the use of `std::ostream_iterator<Fasta>` in the above example is:

```

for(unsigned seq = 0 ; seq < alignment.size() ; ++seq)
{
    out << alignment[seq] << '\n';
}

```

The `for` loop is more familiar to those coming from a C background, but is a more error-prone approach (especially in large programs) than using the STL routines, or so the gurus would have us believe.

### 3.3 A digression: template instantiation in `libsequence`

Warning: this section is technical. Feel free to skip.

In `libsequence`, all template classes and functions are declared and defined in header files. As a method of source code organization, this is often referred to as the "inclusion model" (see Chapter 6 of [9]). This also has

the effect that a rather heavy burden can be placed on compilers during the process of template instantiation. Also, most compilers (including `g++`) use "on-demand instantiation" (discussed in some detail in [9]), meaning that types are only substituted into template code when an instantiation directive occurs in a source file (i.e. a line of code says `std::vector<int>x;`, causing code for the constructor for a `vector<T>` with `T=int` to be generated). On-demand instantiation can have two side effects. One, compile times quickly go through the roof. Two, the size of object code and binaries can get large ("code bloat").

One way to avoid these two problems is to make use of "explicit instantiation", i.e. we tell the compiler what templates we want to use with what types, and get all the code generation over with at once [9]. Unfortunately, the C++ standard doesn't give us an easy way to do that portably [9]. There is, however, a GNU extension to C++ that we can take advantage of if we compile using `g++`. The extension involves using the keyword "`extern`" to declare explicit instantiations, letting us define them in a source file elsewhere. `libsequence` takes advantage of this, providing explicit instantiations for all routines in namespace `Alignment` and the class `Sequence::ClustalW` using type `Sequence::Fasta`. As far as I am aware, this only works on `g++` versions 3 and above. It may work elsewhere, as the `extern` extension is popular, but I haven't tested this or heard any confirming reports. Here's how to use this feature in your code:

```
//check for GNU C++ version 3 and above
#if defined ( __GNUG__ ) && __GNUG__ > 3
//take advantage of explicit instantiations
//at link-time to the library
#include <Sequence/FastaExplicit.hpp>
#else
//need on-demand instantiation
#include <Sequence/Fasta.hpp>
#include <Sequence/Alignment.hpp>
#include <Sequence/Clustalw.hpp>
#endif
```

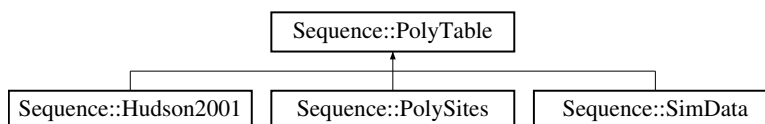
In practice, use of `FastaExplicit.hpp` substantially reduces both compile times and object code size, especially in projects that do data I/O and compile multiple binary executables. The reader can verify for him/herself that substituting the above code into the example in subsection 3.2 results in equivalent functionality.

## 4 Basics of Polymorphism Analysis

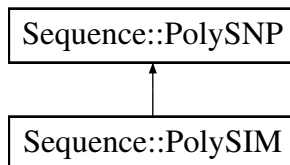
So we're in Section 4 and we haven't learned anything that `bioperl` couldn't already do (and do for 10 times as many sequence and alignment formats). However, the point of `libsequence` is to *analyze* data, and that's where `libsequence` has an advantage. In 4.4, we will write a full-featured program to calculate Tajima's D [8] from a sample of sequence data, and provide command-line options to deal with missing data and sites with more than 2 alleles. In 4.5 we will write code to obtain the expected site frequency spectrum from coalescent simulations. Finally, 4.6 describes code to obtain the distribution of Tajima's D from coalescent simulations.

### 4.1 Inheritance hierarchy

All classes representing polymorphism tables are derived from the virtual base class `Sequence::PolySites`. As a reference for what follows, the inheritance hierarchy for objects representing polymorphism tables is:



The classes that perform *calculations* on objects derived from `Sequence::PolyTable` are:



All polymorphism table objects derived from `Sequence::PolyTable` are compatible with `Sequence::PolySNP`, but the only class compatible with `Sequence::PolySIM` is `Sequence::SimData`.

### 4.2 Acceptable characters and assumptions about phase

`libsequence` gracefully handles the following characters for SNP analysis: A,G,C,T,N,0,1, and -. The '-' is the character used to represent gaps, and

'N' represents missing data. By graceful I mean that characters not in this set are usually just ignored, which isn't graceful. This means that if there are heterozygous sites labeled in the data (which would of course be done using IUPAC characters, R=A/G, etc.), they are ignored. `libsequence`'s classes never ask you if your data are phased or unphased (i.e. if haplotypes are known), but some of the analyses require phased data. The programmer is expected to recognize such cases and handle them as he/she sees fit. For example, allowing the calculation of haplotype diversity on unphased data is a logic error. The reason for this behavior is that, if one considers just a spreadsheet of polymorphic sites, one can't tell if the data are phased or not just by looking, and neither can a program. Its even possible that a sneaky colleague hands you a file of what looks like full sequence data, in which case one normally assumes phases are known. However, such a file may have been generated by taking SNPs from a genotyping project and filling in the invariant sites from a reference sequence.

The header file `Sequence/Alignment.hpp` defines a template function `bool Sequence::Alignment::validForPolyAnalysis(iterator beg, iterator end)`. This function can be used to inspect a range of `std::string`-like objects to make sure that the characters stored behave well with `libsequence`. The function returns `true` if the data are ok, `false` otherwise. The `value_type` of the iterators must be `std::string`, `std::pair<std::string, std::string>`, or something derived from `Sequence::Seq`. This means that you can use this function to inspect a vector (array) of strings, sequence objects, or a polymorphism table object (via the iterators to the individuals, see 4.12).

### 4.3 Accessing data on a PolyTable

The simplest way to think of `Sequence::PolyTable` is as a two-dimensional array of  $n$  rows and  $k$  columns, where  $n$  is the number of individuals/genotypes, and  $k$  the number of segregating sites. `Sequence::PolyTable` supports array-like access, i.e. `data[i][j]` will return the state of the  $j_{th}$  SNP at the  $i_{th}$  individual, provided that  $0 \leq j < k$  and  $0 \leq i < n$ . If  $i$  or  $j$  are out of range, one of two things will occur. The most likely is a segmentation fault. The other possibility is that an assertion will fail, causing a program to exit and print an error message. This second option occurs when `libsequence` is compiled with debugging symbols enabled, in which case range is checked using the simple C function `assert`. The reason for this design was so that we don't have to worry about exceptions for `PolyTable::operator()`, which

would get expensive at run-time for most applications. If your program is segfaulting, linking against a debugging version of the library can help provide a quick diagnostic of what's going wrong.

To ensure that you don't go out of range, the member function `PolyTable::numsites()` returns the number of columns in the data, and `PolyTable::size()` returns the number of rows. For example:

```
PolySites data(...); //we discuss exactly how to
                      //construct this type later on...
for(unsigned j=0 ; j < data.numsites() ; ++j)
{
    for(unsigned i=0 ; i < data.size() ; ++i)
        cout << data[i][j] << endl;
}
```

#### 4.4 Example: calculate Tajima's D from an alignment of DNA sequence data

This example goes full steam ahead. We develop an application with the following specifications:

1. It will output Tajima's D [8] for a data file containing an alignment of sequences in FASTA format. The infile name will be passed to the program with a command-line option, `-i`.
2. If an option, `-n`, is passed, any sites with untyped data (the 'N' character) will be thrown out.
3. If an option, `-b`, is passed, any sites with more than two alleles will not be considered in the analysis.
4. If an option, `-o`, followed by an integer, is passed, the integer shall represent a sequence in the alignment that we will treat as an *outgroup* sequence. By outgroup, we refer to a single sequence from another species. For example, the file may contain 50 sequences from a human gene, and 1 from the same gene in the chimpanzee. The chimp sequence is the outgroup sequence. To be precise, the integer component of the argument will be the *index* of the outgroup sequence, i.e. 0 for the first sequence in the file, etc.

5. The C routine `getopt` will be used to parse command-line arguments.

We make the following simplifying assumptions to keep this within the scope of a tutorial:

1. We assume linux-like behavior concerning `getopt`. This means we assume there is a header, `getopt.h`, in which the function is declared. This is not true in general, as Apple OS X systems and some Sun Solaris boxes (with certain versions of `gcc`) declare `getopt` in `unistd.h`. This is a constant annoyance as far as portability is concerned, and the fix that I use is available from <http://www.molpopgen.org/krthornt>.

The following new concepts are illustrated:

1. Use of class `Sequence::PolySNP`, which contains all the routines for SNP data analysis. This class contains loads of functions, so one should read the reference manual. The use of all of these functions will be similar to this example.
2. Use of class `Sequence::PolySites`, which is an object used to represent the polymorphic sites of an alignment.

While this example seems complicated, we actually already know how to do most of it. We already know how to read in an alignment of FASTA sequences (3.1), and we've seen how to handle exceptions thrown by `libsequence` (3.2). So the only new things to do are to use two new classes from namespace `Sequence` and a widely-used routine to process the command-line.

```
#include <fstream>
#include <iostream>
#include <vector>
#include <getopt.h> //NOT PORTABLE!

//see 3.3 for explanation
//of this preprocessor magic
//check for GNU C++ version 3 and above
#if defined ( __GNUG__ ) && __GNUG__ > 3
//take advantage of explicit instantiations
//at link-time to the library
#include <Sequence/FastaExplicit.hpp>
```

```

#else
//need on-demand instantiation
#include <Sequence/Fasta.hpp>
#include <Sequence/Alignment.hpp>
#endif
#include <Sequence/PolySites.hpp>
#include <Sequence/PolySNP.hpp>
#include <Sequence/SeqExceptions.hpp>

using namespace std;
using namespace Sequence;
using namespace Sequence::Alignment;

//a struct to store command-line arguments
struct params
{
    char * infile;
    bool purgeMissing;
    bool purgeMultiHits;
    bool haveOutgroup;
    unsigned outgroup;
};

//a function to parse the command-line arguments
//see man (3) getopt for the documentation of getopt
int parseargs(int argc, char **argv, params *args)
{
    //establish defaults
    args->infile = NULL;
    args->purgeMissing = false;
    args->purgeMultiHits = false;
    args->haveOutgroup = false;
    args->outgroup = 0;

    extern int optind;
    int c;
    while ((c = getopt (argc, argv, "i:o:nb")) != -1)
    {
        switch (c)
        {

```

```

        case 'i':
            args->infile = optarg;
            break;
        case 'o':
            args->haveOutgroup = true;
            args->outgroup = atoi(optarg);
            break;
        case 'n':
            args->purgeMissing = true;
            break;
        case 'b':
            args->purgeMultiHits = true;
            break;
        default:
            cerr << "Huh?" << endl;
            exit(1);
            break;
    }
}
return optind;
}

//get to work
int main(int argc, char **argv)
{
    params args;
    parseargs(argc,argv,&args);
    if (args.infile == NULL)
    {
        cerr << "error: no infile specified!\n";
        exit(1);
    }
    vector<Fasta> alignment;
    try
    {
        GetData(alignment,args.infile);
    }
    catch(SeqException &e)
    {
        cerr << e << endl;
    }
}

```

```

        exit(1);
    }
    if (args.outgroup >= alignment.size())
    {
        cerr << "error:  outgroup sequence out of range!\n";
        exit(1);
    }
    if (! IsAlignment(alignment) )
    {
        cerr << "error:  data not aligned!\n";
        exit(1);
    }
    //now, generate a polymorphism table
    //the command-line args get applied here in the constructor call
    //see reference manual for details
    //when used, the third argument must always be "true"
    PolySites SNPtable(alignment,args.purgeMultihits,true,
        args.purgeMissing);

    //now, create an object of type Sequence::PolySNP
    //This object knows how to calculate Tajima's D
    //The other command-line arguments let this
    //object know how to deal with the outgroup.
    //The second constructor argument is a boolean, which
    //is true if there is an outgroup, false otherwise,
    //which is the default.
    //The third argument is the index of the outgroup
    //in the data vector (defaults to 0).
    PolySNP calculator(&SNPtable,args.haveOutgroup,args.outgroup);

    //now, do the calculation...
    cout << "Tajima's D for file << args.infile
        << " is:  " << calculator.TajimasD() << endl;
    return 0;
}

```

This example accomplished the following:

1. Handled all I/O and performed error-checking of input data file.
2. Allowed the treatment of the data with respect to missing data and

multiple hits to be specified on the command-line by the user

3. Generate a polymorphism table and did calculations on it.
4. Took advantage of explicit instantiation of templates on platforms where supported

#### 4.4.1 Comments on outgroup sequences

We just saw that the constructor of `Sequence::PolySNP` takes arguments telling the class whether or not outgroup information is present in the data. There is only space for one outgroup. In practice, several sequences may be collected from outgroup taxa. In such cases, the relevant information is the *inferred ancestral state* of a position in the ingroup data. For such cases, the "outgroup" sequence can be considered as a list of inferred ancestral states.

#### 4.4.2 An important caveat

As we just saw, `Sequence::PolySNP` can take a `const Sequence::PolySites *` as a constructor argument (actually, it takes a `const Sequence::PolyTable *`, which is the base class of `PolySites`, but that doesn't matter now). Internally, `PolySNP` assigns a `const PolyTable *` to point to your data. This is important, because it means that *no new object representing your data is allocated*. The reason for this is efficiency—we avoid copying by doing this (which means both less memory used and fewer constructor/destructor calls made). However, if your original `PolySites` object is deleted or goes out of scope before the destructor of the `PolySNP` object is called, any further use of `PolySNP` member functions results in undefined behavior. In other words, the following code snippet is just plain wrong:

```
vector<Fasta> data; //our alignment, as before
GetData(data,infile); //fill alignment from a file
PolySites *table = new PolySites(data); //make SNP table
PolySNP calculator(table); //make object to do calculations
//now, calculator contains a const pointer to table
delete table; //now, that pointer points to a deleted object!
cout << calculator.TajimasD() << endl; //undefined...
```

## 4.5 Example: calculate the frequency spectrum from coalescent simulation data

In 4.4, we learned how to do calculations on polymorphism data. However, we restricted ourselves to calculations already coded into `libsequence`. In this example, we will do calculations that `libsequence` doesn't already have a routine for. Our application will be to read in data from Dick Hudson's coalescent simulation program `ms` [3], which is available from <http://home.uchicago.edu/~rhudson1>. For each simulated data set, we will calculate the site frequency spectrum and print it to `stdout`. The site frequency spectrum is defined as the number of times a mutation is observed once, twice, etc. in the sample. More info on coalescent theory and simulation is available in [1, 2].

Before we continue, we need to explain something. The class `Sequence::PolySites` that we used in 4.4 is derived from an abstract base class, `Sequence::PolyTable`. Objects derived from this base class can be acted upon as if they are two-dimensional arrays, where the rows are sequences (individuals) and the columns the segregating sites. The member function `size()` returns the number of sequences, and `numsites()` the number of segregating sites (the numbers are returned as `unsigned` values).

`libsequence` defines a class `Sequence::SimData`, derived from `Sequence::PolyTable` that can be used to read in data from Hudson's program, `ms`. This class provides two ways to read in data from `ms`. This first is using `operator>>`, as expected. However, `operator>>` can be much slower than using the C routines in `<cstdio>`, so a member function, `int fromfile(FILE *)` is provided to read routines from *already-opened* C-style file pointers. This function can also be used to read from `stdin` (i.e. from a Unix pipe in most cases) using C I/O functions.

Before `ms` outputs any simulated samples, it outputs the command-line that was input. Therefore, one must use `Sequence::SimParams` to read in those parameters using `operator>>`. `Sequence::SimParams` defines a function, `int fromfile( FILE * )` as described above.

If you are not familiar with `ms`, the data simulated are labeled with 0's and 1's at each polymorphic site in the data. The 0 represents the ancestral state at that site, and the 1 represents a derived mutation. In the following example, we will count the number of 1's at each polymorphic position. So what we are actually doing is calculating the frequency spectrum for *derived* mutations (sometimes also called polarized mutations).

```

#include <Sequence/SimData.hpp>
#include <Sequence/SimParams.hpp>
#include <cstdlib>
#include <cstdio>
#include <vector>

//msfreq
//Kevin Thornton, k-thornton@uchicago.edu

//to compile:
//g++ -o msfreq msfreq.cc -lsequence -O3

//example usage:
//ms 10 10 -t 10 -r 10 10 | ./msfreq

using namespace std;
//Sequence is the namespace (scope) defined in libsequence
using namespace Sequence;
int main(int argc, char *argv[])
{

    //an object of type Sequence::SimParams
    SimParams p;
    p.fromfile(stdin);
    //an object of type Sequence::SimData
    SimData data;
    //sync the cin stream with fscanf
    //(requires very ANSI compliant c++ compiler)
    std::ios_base::sync_with_stdio(true);
    int rv;

    while( (rv = data.fromfile(stdin)) != EOF )
    {
        //store the freq spectrum as a vector of unsigned values
        //for a sample of n simulated gametes, we initialize
        //freqSpec with n-1 0's
        vector<unsigned> freqSpec(data.size()-1,0u);
        //iterate over each polymorphic site
        for(unsigned site = 0 ; site < data.numsites() ; ++site)

```

```

    {
        unsigned nc = 0; //"number of changes"
        //iterate over each sequence in the data
        for(unsigned seq = 0 ; seq < data.size() ; ++seq)
        {
            //note how an object of type Sequence::SimData
            //acts like a 2D array
            //Also, the data in objects derived from
            //Sequence::PolyTable are stored in a
            //std::vector<std::string>, so we need to
            //check for 1 as a char rather than an int
            nc += (data[seq][site] == '1') ? 1 : 0;
        }
        freqSpec[nc-1]++;
    }

    //print output, and use fprintf for speed
    for(unsigned freq = 0 ; freq < freqSpec.size() ; ++freq)
    {
        fprintf(stdout,"%d %d\n",freq+1,freqSpec[freq]);
    }
    fprintf(stdout,"/>\n");
}
return 0;
}

```

## 4.6 Example: obtain the distribution of Tajima's D from coalescent simulation

Similar to 4.4, we can also calculate Tajima's D (and lots of other things) on objects of type `Sequence::SimData`. However, data from coalescent simulation is of a much simpler sort than sequence data. There are only 2 characters (0 and 1), no missing data, etc. `libsequence` provides `Sequence::PolySIM`, which is derived from the class `Sequence::PolySNP` that we used in 4.4, but with the implementation optimized for the simpler data. Rather than write a whole program, we will show a code snippet illustrating how to calculate Tajima's D using `Sequence::PolySIM`.

```
#include <Sequence/PolySIM.hpp>
```

```

Sequence::SimParams p;
cin >> p;
Sequence::SimData data(p.totsam());
//start reading in the data as in 4.5
Sequence::PolySIM calc(&data);
cout << calc.TajimasD() << endl;

```

By substituting the above snippet into the example of 4.5 (doing so will be left as an exercise for you), you've got a program that outputs the distribution of Tajima's D to `stdout`.

## 4.7 Counting the states at a site in a polymorphism table

Many calculations in population genetics do basically the same thing—march site by site through a polymorphism table and count something. This gets tedious to code. The header `Sequence/stateCounter.hpp` describes a *function object*, `Sequence::stateCounter` that simplifies this task. Here is its declaration:

```

class stateCounter : public std::unary_function<char,void>
{
private:
    mutable char _gap;
public:
    mutable unsigned a,g,c,t,zero,one,gap,n;
    mutable bool ndna;
    stateCounter(const char &gapchar = '-');
    void operator()(const char &ch) const;
    unsigned nStates(void) const;
};

```

The `mutable unsigned` values store the number of A,G,C,T,0,1,N, and gaps ('-') that were encountered. Since they are public, they are accessible directly, although not modifiable without a `const_cast`. The boolean `ndna` is true if a character other than those just listed were encountered, false otherwise. The function `nStates()` returns the number of states at the site. Here's an example inferring the number of mutations at each site, skipping sites with gaps and non-DNA characters. For a site with  $k$  alleles, there are

$k - 1$  mutations inferred:

```
vector<Fasta> data;
//fill data..
PolySites table(data);
unsigned nmutts = 0;
for(unsigned site = 0 ; site < table.numsites() ; ++site)
{
    stateCounter counts;
    for(unsigned seq = 0 ; seq < table.size() ; ++seq)
    {
        counts(table[seq][site]);
    }
    if (counts.nStates() >= 2 && counts.gap == 0
        && counts.ndna == false)
    {
        nmutts += counts.nStates()-1;
    }
}
```

There is another clever use of `Sequence::stateCounter`. The following snippet calculates the base composition of a sequence using the STL algorithm `std::for_each` from `<algorithm>`:

```
Sequence::Fasta seq;
Sequence::stateCounter baseComp = std::for_each(seq.begin(),
        seq.end(), Sequence::stateCounter('-'));
```

The object `baseComp` now knows how many A,G,C,T,N, and gap characters are present in `seq`, as well as whether or not any non-DNA characters were encountered. Since `std::for_each` operates over a range, it is easy to extend the above example to calculate base compositions over a sliding window along a sequence:

```
Sequence::Fasta seq;
vector<string>::iterator beg = seq.beg();
unsigned windowLen = 10;
unsigned jumpSize = 10;
while (beg + windowLen < seq.end())
{
```

```

    stateCounter baseComp = std::for_each(beg,
        beg + windowLen, stateCounter('-'));
    //do something with baseComp here....
    //adjust the window
    beg = (beg + jumpSize < seq.end())
        ? beg+jumpSize : seq.end();
}

```

A close read of the above example reveals that small windows at the end of the sequence will be skipped if they are less than either `windowLen` or `jumpSize` in length.

#### 4.7.1 A more general counter

The functor `Sequence::stateCounter` is intended for the analysis of nucleotide data with a relatively small alphabet (A,G,C,T,N). It is sometimes useful to have a more general counter, and the function `Sequence::makeCountList` in the header `Sequence/SeqUtilities.hpp` provides one. The function's declaration is:

```

template<typename Iterator>
typename std::map<typename std::iterator_traits<Iterator>::value_type, unsigned>
makeCountList( Iterator beg, Iterator end);

```

Deciphering the STL-ese reveals that the return type is a `std::map` containing the type present in the range `(beg,end]` and an `unsigned` integer. In essence, the function returns a map of each element in the range and the number of times it occurred. To use this with polymorphism table objects, one would have to transpose the data matrix such that each column (i.e. site) could be accessed via iterators. How to do so is discussed in 4.14 on page 41 below.

## 4.8 One last class: SimpleSNP

There is one more class representing polymorphism tables, `Sequence::SimpleSNP`. It is declared in `Sequence/SimpleSNP.hpp`. The operators `operator<<` and `operator>>` are defined for I/O. The format of the tables is as follows. The first line consists of the numbers of sequences and SNPs. The second line is a list of SNP positions. The third line is a list of ancestral states for each SNP. If the ancestral state is unknown, this line is a list of 'N' characters. The

rest of the table consists of a label for each individual and the states for each SNP. Missing data are represented by the 'N' character. Here's an example for two sites in sample size 4, with the outgroup state unknown:

```
4      2
N      N
ind1   A  C
ind2   A  G
ind3   C  G
ind4   A  C
```

This class used to be called `Sequence::Hudson2001`, and a typedef is provided to keep that old name valid.

## 4.9 Other member functions of polymorphism table classes

The class `Sequence::PolyTable` imparts a fair amount of functionality to derived classes. We briefly discuss these functions here. Note that many of the function prototypes below employ default arguments. These represent common situation for which one may use such functions, so it is often the case that one need not pass any arguments at all. If you're not familiar with this aspect of C++, consult Stroustrup [7].

### 4.9.1 Obtaining the "raw" data

Internally, the polymorphic sites are stored in a `std::vector<std::string>`, and the positions of each site in a `std::vector<double>`. The member function `GetData()` returns the `std::vector<std::string>`, and `GetPositions()` returns the `std::vector<double>`. There are also iterators to both the data and the positions, as well as `const_iterators`. They are specified by the following typedefs:

```
typedef std::vector<string>::iterator data_iterator;
typedef std::vector<string>::const_iterator const_data_iterator;
typedef std::vector<double>::iterator pos_iterator;
typedef std::vector<double>::const_iterator const_pos_iterator;
```

The iterators are returned by the following member functions, which are named to be similar to the familiar `begin()` and `end()` of STL containers:

```

data_iterator begin();
data_iterator end();
const_data_iterator begin() const;
const_data_iterator end() const;
pos_iterator pbegin();
pos_iterator pend();
const_pos_iterator pbegin() const;
const_pos_iterator pend() const;

```

Iterators are discussed in more detail below in 4.10, including a description of a special iterator that allows iteration over segregating sites in the SNP table (4.13, p. 38).

#### 4.9.2 Accessing the positions of segregating sites

The member function `position(unsigned i)` returns the position of the  $i_{th}$  segregating site as a `double`. The value returned depends on how the polymorphism table object was originally created (normally it would be the actual positions in the alignment). The member function `numsites()`, which returns the number of positions in the alignment, can be used to establish the termination condition of `for` loops over the list of positions. Alternatively, the iterators described above can be used in the standard fashion.

#### 4.9.3 Removing sites with more than two alleles

The following member function:

```

virtual void RemoveMultiHits(bool skipOutgroup=false,
                             unsigned outgroup=0);

```

removes all sites from the data with more than 2 states present. If the argument `skipOutgroup` is `true`, then the character state of the outgroup does not count towards the number of states at the site. The index of the outgroup in the data can be specified by passing an `unsigned` value as the second argument.

#### 4.9.4 Removing missing data from a polymorphism table

In `libsequence`, the 'N' character is considered to be missing data for DNA sequences. Missing data is a fact of life in population genetics these days. While the routines in `libsequence` handle missing data in a reasonable way

(see the documentation of `Sequence::PolySNP` in reference manual for details). However, if you make a table with missing data, you may want to remove it later. This member function does that for you:

```
virtual void RemoveMissing(bool skipOutgroup=false,
                          unsigned outgroup=0);
```

The arguments are the same as described in 4.9.3 above.

#### 4.9.5 Removing sites below a certain frequency

To do this, use the following member function:

```
virtual void ApplyFreqFilter(unsigned mincount,
                             bool haveOutgroup = false,
                             unsigned outgroup = 0);
```

The argument `mincount` refers to the *minimum number of times an allele must be present in the sample*, else that site is purged from the table. When `haveOutgroup == false`, the frequency of the *minor* allele is what's relevant, and the minor allele must occur at least `mincount` times, else the site is purged. When `haveOutgroup == true`, the frequency of the *derived* allele is what's relevant, and *all derived alleles must occur more than mincount times*, else they are purged. (That last qualification may seem strange, but it's important for when there are more than 2 alleles at a site).

#### 4.9.6 Converting the data into a binary format (and using objects derived from `Sequence::PolyTable` objects with C code)

You may already have routines that assume data are encoded as 0 and 1. The following routine will convert polymorphism tables generated from sequence data to 0's and 1's:

```
virtual void Binary (bool haveOutgroup = false,
                    unsigned outgroup = 0,
                    bool strictInfSites = true);
```

If `haveOutgroup == false`, the 0 and 1 are assigned arbitrarily, else 0 represent ancestral and 1 derived. All sites with more than 2 alleles are purged (and the outgroup state always counts towards the total number of states). The third argument is deprecated and has no effect, and will be removed in a later release.

I personally do not find this function all that useful, but in combination

with the ability to access the data underlying `Sequence::PolyTable` (4.9.1 above), and using the tips in described in [5] on passing C++ containers to C APIs, you can actually manipulate polymorphism data in a way that's compatible with any existing C code you have lying around.

#### 4.9.7 Assigning data to a PolyTable

The following template member is provided for assignment of data to a `PolyTable`:

```
template<typename numeric_type, typename string_type>
bool assign( const numeric_type * _positions,
             const unsigned & _num_positions,
             const string_type * _data,
             const unsigned & _num_individuals );
```

The function returns `true` if assignment was successful, `false` otherwise, in which case you will be left with an object containing no data. Assignment will be unsuccessful if the length of the strings is not equal to `_num_positions`.

An additional function allows assignment from a range of `PolyTable::const_site_iterator` (see 4.13, p. 38 below):

```
bool assign(PolyTable::const_site_iterator beg,
            PolyTable::const_site_iterator end);
```

## 4.10 Iterators for polymorphism table classes

### 4.10.1 A digression—how to represent SNP data?

One has a data set of  $k$  segregating sites in a sample of  $n$  individuals. The data are to be read into a program, and some statistic of interest calculated. It seems clear that the data should be stored in a  $k$ -by- $n$  array, but should it be  $k$  columns by  $n$  rows, or vice-versa? Depending on the application, one representation may be more convenient than the other. `Sequence::PolyTable` (and all classes derived from it) support both types of representation *simultaneously*. This makes the class extremely flexible in terms of algorithm design, at the cost that certain types of operations may be expensive at run-time. This affects how the iterators for this class are implemented, as will be explained below in 4.13.

## 4.11 Iterators to site positions

Internally, the positions of segregating sites are stored in a `std::vector<double>`. The member functions `PolyTable::pbegin()` and `PolyTable::pend()` provide both `const` and non-`const` access to this vector.

## 4.12 Iterators to individuals

`PolyTable::operator[] (unsigned i)` offers the array access that one would expect, returning a `std::string` representing the states of all SNPs for the  $i_{th}$  individual. As you may guess, the data are stored in a `std::vector<std::string>` of size  $n$  to represent the  $n$  individuals in the sample. This vector can be iterated over using the member functions `PolyTable::begin()` and `PolyTable::end()`. These functions can be used in both `const` and non-`const` contexts.

## 4.13 Iterators to segregating sites

This is the most useful iterator, and also potentially the easiest to get yourself in trouble with. As mentioned above in the previous 2 subsections, the data representing the SNP positions and the states are stored in 2 separate vectors. This provides natural ways to implement iterators to positions and individuals, however, it is also useful to iterate over *segregating sites*. It can be done, using member functions `PolyTable::sbegin()` and `PolyTable::send()`. When these functions are called, a new representation of the data is created, a `std::vector< std::pair<double, std::string> >`. There is one element in the data for each SNP, and the double represents the position, and the string the states at that site. Along the string, the states are in order of the 0 to  $n-1$  individual that `Polytable::operator[]` would access. The creation of this vector uses the machinery described below (4.14).

Iterators to segregating sites are of type `PolyTable::const_site_iterator`, which is a typedef for `std::vector<std::pair<double, std::string> >::const_iterator`, and can only be used in `const` contexts. Note that the value type of these iterators is a `std::pair<double, std::string>` (I use value type in the STL sense of what you get when you de-reference an iterator), i.e.

```
vector<Fasta> alignment;
//fill alignment...
PolySites SNPs(alignment);
PolySites::const_site_iterator beg = SNPs.sbegin();
```

```

//as with any pair, first is the first element, and second the second
cout << beg->first //print out position
    << ' '
    << beg->second //print list of states at this site
    << endl;

```

#### 4.13.1 Why this can be an expensive iterator

This is a useful iterator, but there is a cost. It requires a representation of the data that is not the default representation used to represent the class data. Thus, when these iterators are accessed, this class must check if it needs to create the new representation. In fact, *any time a non-const member function is called, the functions `PolyTable::sbegin()` and `PolyTable::send()` will re-calculate the `std::vector< std::pair<double, std::string> >!`* This should not discourage use of these functions, as the cost of calculating this "rotation" of the data is often negligible, but being aware of this potential issue will help you avoid doing a "negligible" calculation 100's of times, at which point it may add up at run-time.

#### 4.13.2 Don't invalidate your pointers!

Iterators are cool, but they can become "invalid". Pointer invalidation refers to the assignment of pointer followed by an action resulting in that pointer either pointing at nothing or not pointing at what the programmer thinks it's pointing at. For example, let's say you are implementing a linked list as follows, `std::vector< node * >`, where `node` is a class/struct of your own design that points to the next element in the list. The following code seems innocent:

```

vector<node *> nodes;
//do some stuff
node *new_node(...);//construct new node
nodes.push_back(new_node);

```

Odd are, this will compile fine, but at run-time, it will not do what you expect, for the following reason. The `node *` in `nodes` point *at each other*, and when `std::vector` needs to reallocate when `push_back` is called, *it will likely move the data to a new place*, meaning who knows what your nodes now point at. In other words, they're invalid.

Pointer invalidation can also occur using `PolyTable::const_site_iterators`.

For example:

```
vector<Fasta> alignment;
//fill alignment...
PolySites SNPs(alignment);
PolySites::const_site_iterator beg = SNPs.sbegin();
SNPs.RemoveMissing(); //remove SNPs with missing data
//as with any pair, first is the first element, and second the second
while(beg < SNPs.send())
{
    cout << beg->first //print out position
        << ' '
        << beg->second //print list of states at this site
        << endl;
    ++beg;
}
```

The behavior of the above snippet is undefined. The explanation is that, after initialization of `beg`, the non-const member function `PolyTable::RemoveMissing` was called. That sent a signal to the class that the representation of the data from which `const_site_iterators` are drawn needs recalculation. That recalculation occurs *in the while loop when `PolyTable::send()` is called!*. In other words, `beg` and the new `send()` are likely not in a contiguous memory space, and that's bad.

A correct implementation would have been:

```
vector<Fasta> alignment;
//fill alignment...
PolySites SNPs(alignment);
SNPs.RemoveMissing(); //remove SNPs with missing data
//declare the iterators after manipulating the object
PolySites::const_site_iterator beg = SNPs.sbegin(),
    end = SNPs.send();
//as with any pair, first is the first element, and second the second
while(beg < SNPs.send())
{
    cout << beg->first //print out position
        << ' '
        << beg->second //print list of states at this site
        << endl;
    ++beg;
}
```

```
}
```

In this version, `beg` and `end` define a valid contiguous range.

#### 4.14 "Rotating" a SNP table

Classes derived from `Sequence::PolyTable` are two-dimensional arrays where the rows represent individuals and the columns character states at each polymorphic site. For some calculations, this is inconvenient, and one may want the table to be "rotated". The header `PolyTableManip.hpp` declares functions to do these rotations. This aspect of `libsequence` is under development, and future releases will contain more functions. As of `libsequence 1.3.1`, only one function is declared:

```
Sequence::Locus Sequence::makeLocus(const Sequence::PolyTable *table);
```

The type `Sequence::Locus` is a typedef for `std::vector< Sequence::Site>`, and `Sequence::Site` is a typedef for `std::pair<double,string>`. In other words, this function returns a list of pairs. For each pair, the first element is the position, and the second is the raw data for that site (the order of the raw data are not changed, so from left to right along the *string* is the same as it was from top to bottom of the original `PolyTable`. The first character at each site represent the first individual in the data (i.e. the order doesn't change). `Sequence::Locus` is the "rotation" of polymorphism tables that is sometimes useful. Future releases will contain functions to rotate `Sequence::Locus`, returning a `Sequence::PolySites`, and there will likely be a function that will process a `Sequence::Site`, returning a `std::vector<char>` containing a list of characters at the `Site`.

One should note that transposing the data matrix like this makes available other methods to tasks like counting the numbers of states per site. Let's revisit the example use of `Sequence::stateCounter` from 4.7 above. The point of the following snippet is just to show that the transposition allows the use of `std::for_each` to count the number of states per site:

```
vector<Fasta> data;  
//fill data..  
PolySites table(data);  
Locus l = makeLocus(&data);  
for(unsigned i = 0;i<Locus.size();++i)  
{
```

```

    stateCounter s = for_each(l.second.begin(),
        l.second.end(),stateCounter('-'));
    //do something with s
}

```

Alternatively, if we didn't want to restrict ourselves to the character set {A,G,C,T,N,0,1}, we could substitute `std::set`, from `<set>` into the above example:

```

vector<Fasta> data;
//fill data..
PolySites table(data);
Locus l = makeLocus(&data);
for(unsigned i = 0;i<Locus.size();++i)
{
    std::set s(l.second.begin(),
        l.second.end());
    //do something with s
}

```

This last example can be quite useful, as the `std::set` container can be quite useful for tasks such as calculating whether or not different partitions of an alignment share the same characters at a particular site. See [7] for a discussion of `std::set`. Further, the transposition of the data to type `Sequence::Locus` makes it easier to use range operations and algorithms on a subset of the data. For example, where we used `Sequence::stateCounter` in combination with `std::for_each`, we could just have easily only counted the states from the  $i_{th}$  to the  $j_{th}$  characters in the alignment using pointer arithmetic:

```

stateCounter s = for_each(l.second.begin()+i,
    l.second.begin()+j,stateCounter('-'));

```

The function `Sequence::makeLocus` is used internally in `libsequence` when calculating linkage disequilibrium statistics. For such calculations, I found it convenient to have the data "rotated", which gave rise to this header. Since then, I have found repeated need for a function returning a list of states present at each site, but that code hasn't been ported back to the library yet.

## 4.15 The header `Sequence/PolyTableFunctions.hpp`

There are lots of operations that one may want to do to manipulate or rearrange data in a polymorphism table. Originally, such functions were added as members of the base class, `Sequence::PolyTable`, but that becomes unwieldy as it breaks binary compatibility every time a new function is added. The policy now is to declare these functions in the header `Sequence/PolyTableFunctions.hpp`. See the reference manual (Section 1.8, p. 7) for the documentation of these functions.

## 5 Counting

Much of computational biology is counting. In section 4.7 (p. 31), we discussed how to keep track of counts of alleles at a site in a polymorphism table. However, `libsequence` contains a few other features to simplify counting operations. One of these has already been mentioned, `Sequence::makeCountList` (section 4.7.1, p. 33), which returns a `std::map< type, unsigned >` that stores the counts of all the elements of type `type` in some range.

### 5.1 Sequence/CountingOperators.hpp

`libsequence` provides a header, `<Sequence/CountingOperators.hpp>`, that defines the following templates:

```
template<typename key, typename value>
std::vector<std::pair<key,value> >
operator+=( std::vector<std::pair<key,value> > &lhs,
const std::vector<std::pair<key,value> > &rhs);

template< typename key, typename value,
typename comparison>
std::map<key,value,comparison> operator+(
const std::map<key,value,comparison> &lhs,
const std::map<key,value,comparison> &rhs);

template< typename key, typename value, typename comparison>
std::map<key,value,comparison> operator+=(
std::map<key,value,comparison> &lhs,
const std::map<key,value,comparison> &rhs);
```

These operators allow the return value of `Sequence::makeCountList` to be added together using the standard `+` and `+=` operations. Note that they do so by copying, which can be expensive for some types. In order to compile, type `key` must be comparable using `operator==`, and type `value` must have operators `+` and `+=` defined.

## References

- [1] R. R. Hudson. Properties of a neutral allele model with intragenic recombination. *Theoretical Population Biology*, 23(2):183–201, 1983.
- [2] R. R. Hudson. Gene genealogies and the coalescent process. In *Oxford Surveys in Evolutionary Biology*, volume 7, pages 1–42. 1990.
- [3] R. R. Hudson. Generating samples under a wright-fisher neutral model of genetic variation. *Bioinformatics*, 18:337–338, 2002.
- [4] S. Meyers. *Effective C++*. Addison-Wesley Publishing Company, 1996.
- [5] S. Meyers. *More Effective C++*. Addison-Wesley Publishing Company, 1998.
- [6] S. Meyers. *Effective STL*. Addison-Wesley Publishing Company, 2001.
- [7] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, 3rd edition, 1997.
- [8] F. Tajima. Statistical-method for testing the neutral mutation hypothesis by dna polymorphism. *Genetics*, 123(3):585–595, 1989.
- [9] D. Vandevoorde and N.M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley Publishing Company, 2003.